CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Pascal Library

Mary R. Thompson
Sharon Schanzer
Dean Zarras

23 August 1984
Spice Document S169

Abstract

This document will eventually provide documentation of all the Pascal library routines. Much of the information in this section has been taken from comments in the modules and is no more accurate than those comments.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

This manual includes sections from the PERQ Systems Manual, *The Pascal Library*.

Table of Contents

Acknowledgments

The routines in the Pascal Library have been written by many different people over the past several years. The names of the major implementers for each module are included in the module sections.

Where to find things

All Pascal Library software is currently stored on the CMU-CS-CFS Host (a Vax Unix) in a directory tree rooted at **/usr/spice/libpascal**. To get the run files for up-to-date Spice system, including both the Accent kernel, the servers, and the standard utility programs, use the **Update** program, type

```
path /Sys/Spice
```

This places you in the proper directory. Depending on what type of PERQ you are using, type

```
update perq1a_OIO
update perq1a_CIO
update perq2
update perqt2
```

See the *Introduction to the Spice User's Manual* or *Update: A File Transfer Facility* for instructions on how to determine the type of PERQ you are using.

If you are going to be writing programs that use the Pascal Library routines you will need the Exports sections of the source files. The standard way to do this is to create a subdirctory **/Sys/Spice/LibPascal** on your Perq and then type

```
path /Sys/Spice/LibPascal
update stub   get Exports sections of source files
```

To get the longer version of the Pascal Library including comments, type

```
update longstub
```

Problem reports

All reports of problems related to Spice or its documentation should be mailed to the ArpaNet address **Spice@CMU-CS-Spice**. This address may be abbreviated to **Spice@Spice** on CMU Computer Science Department computers. To keep track of the latest changes in systems, documentation, and procedures, all Spice users should read the "Spice" bulletin board on any of the local host-machines.

Due to the number of routines documented here and the fact that much of the descriptive information was taken from program comments, there may be errors and ommisions in the explanatory material. The calling sequences were copied from the code, so they should be correct. Please report any errors that you notice to **Spice@Spice**, and every attempt will be made to correct the document.

Anyone who modifies any of these modules or adds new modules to the LibPascal directory is *strongly* urged to notify **Spice@spice** attn:documentation group of any changes. Do not assume that simply changing a standard program will ensure that the documentation elves will know about it.

1 Introduction

This document attempts to describe all the generally useful subroutines that are in the Pascal Library. To be more specific, these are the programs that can be found in /usr/spice/libpascal on the CFS vax or boot:libpascal on a Perq. These programs can be organized into three groups:

1. General utility routines:

aload	bootinfo	cload
clock	commandparse	configuration
extracmdparse	ipcrecordio	oldtimestamp
pathname	pmatch	realfunctions
rundefs	salterror	spawn
spice_string	viewkernel	windowutils

2. Routines that provide remote procedure call interfaces to servers:

acccall	accint	auth
envmgr	etheruser	io
modgetevent	msgn	procmgr
sapph	sesame	time
ts	viewpt	

3. Routines that provide runtime support for Pascal code:

dynamic	except	pascalinit
paslong	pasreal	reader
stream	writer	

The next section of this document provides a brief abstract of each of these modules and tells where it is documented more completely. Following that there is a section for most of the modules that have not already been documented elsewhere.

Each module section contains a chronological list of the implementors that was taken from the ChangeLog in the program. The first name on the list is the person who did the original implementation, while the last name is the person who has most recently modified it. Each section also contains a list of files. The first file is the Pascal code for the module. The other files contains definitions of types, constants or variables that are used by the exported procedures of this module. In addition to the files listed, each module also uses accenttype.pas to provide the definition of the type *port*.

Next, each section includes the definitions of all the types used in calls to the module and all the exported exceptions. If any type or values are missing you can look in the section **Accent Public Module Index** of the **Spice Programmers' Manual** which provides an alphabetical list of all the names exported by any module in the Pascal library directory. Each entry in that section gives the definition of the item and the name of the module that exports it.

Finally, each section provides a description of all the routines exported by the module.

The directory /usr/spice/devices is one more source of low-level utility routines. This directory contains modules that deal directly with devices or machine configurations. These are routines that must be run with privileges enabled.

2 Module Summary

The following is a list of the modules in the **LibPascal** directory. The ones marked with an '*' are described in this document. If a module is not documented we suggest that you look at the code for more information.

AccCall: Accent Kernel non-message interface. Documented in the **Accent Kernel Interface Manual.**

AccInt Accent Kernel message interface. Documented in the **Accent Kernel Interface Manual.**

***ALoad** Pascal loader. Also provides a routine to display run files.

Auth Provides the user interface to the authorization server. Documented in the **Servers Manual**

***BootInfo** Provides the definition of the Boot Information Block.

***CLoad** Provides the facilities to load a process with a C program.

***Clock** Provides a quick 60-hz clock routine.

Code Provides common definitions for the linker and loader. This module is not documented.

***CommandDefs** Provides definitions for the command structure passed between Accent programs.

***CommandParse** Provides routines to parse and interpret a command line.

***Configuration** Provides information about the hardware configuration of the current machine. To use this module a process must have access to physical memory.

***Dynamic** Pascal dynamic storage routines. Generally calls to these routines are only generated by the Pascal compiler.

***EnvMgr** User interface to the Environment Manager. Provides routines to set and read environment variables. Documented in the **Servers Manual.**

***ExtraCmdParse** More routines to parse command lines and to answer yes-or-no questions.

EtherUser User interface to the Ether Server. Provides routines to do 'direct' ether I/O.

***Except** Pascal runtime support for exceptions. This module exports many of the Pascal standard exceptions. These are documented here. The routines in this module are not called by the user and are not documented.

***IPCRecordIO** Provides routines to send and receive a simple record.

ModeGetEvent Provides routines to do Sapphire functions asychronously. Not documented here. Documented in the **Saphire Window Manager Procedure Headers.**

MsgN This module is the interface to the Message-Name Server. Provides routines to register and lookup Ports in a net-work-wide name table. Documented in the **Servers Manual**

NameErrors This module exports the MsgServer (NameServer) errors. It is not documented.

***OldTimeStamp** Provides routines to translate between POS and Accent date formats.

***PascalInit** This is the module called immediately after process creation to initialize a process to run Pascal code and to communicate with other processes. Users do not call this module, but it exports many of the ports to the standard servers. Its exported variables, and exceptions are documented here.

PasLong Pascal runtime procedures to convert longs for input and output. These routines are not called by users, thus they are not documented.

PasReal Pascal runtime procedures to convert reals for input and output. These routines are not called by users, thus they are not documented.

***PathName** High level interface to the Sesame File Server. Uses logical names in addition to absolute pathnames.

***PMatch** Provides routines for pattern matching in strings.

ProcMgr This module is the interface to the Process Manager. Provides routines to register and manipulate processes. Documented in the **Servers Manual.**

QMapDefs This module defines the constants and types used by the Qcode to source mapping facility. It is not documented.

Reader Pascal runtime support for input. These routines are not called by users, thus are not documented.

***RealFunctions** Provides routines to evaluate many standard logarithmic and trigonometric functions on real numbers.

RunDefs This module defines the form of Pascal .Run files. This module is not documented.

***SaltError** Provides a routine to return a string explaining the meaning of a standard system error code.

Sapph This module is the interface to the Sapphire Window Manager. It is documented in the section **Sapphire Window Manager Procedure Headers.**

SegDefs This module defines the constants and types used in the .seg file output of the Perq Pascal compiler. It is not documented.

Sesame	This module is the interface to the Sesamoid File Server. The Pascal calling sequences are documented in the **Servers Manual**, but the explanation of the file system is in **Sesame: The Spice File Server.**
***Spawn**	Provides routines for process creation and initialization.
***Spice_String**	Provides many routines for hacking Pascal strings.
***Stream**	Lowest level runtime support for Pascal I/O. The routines of this module are not called by users, but all the standard Pascal I/O exceptions are exported from here..
SymDefs	This module contains the definitions for the symbol table file produced by the current Pascal compiler. This module is not documented.
Time	This module is the inteface to the Time Server. Provides routines to get the current date and time and to translate between the various formats for dates and time. Documented in the **Servers Manual.**
TS	This module is the interface to the Typescript Server. Provides routines to create and manipulate Typescripts which are used for character steam I/O into a window. Documented in the **Servers Manual.**
***ViewKernel**	Routines to do graphic I/O to windows. Documented here and in the section **Sapphire Window Manager Procedure Headers.**
ViewPt	This module is the interface to the ViewPort server. It is documented in the section **Sapphire Window Manager Procedure Headers.**
***WindowUtils**	Provides routines to change the title line of the UserWindow.
Writer	Runtime support routines for Pascal output. Routines are not generally called by users, never-the-less it is documented here.

3 Aload: A pascal process loader for accent

Implementers: J. Eugene Ball
Michael Jones
Doug Philips
Richard Gumpertz

Abstract: Aload provides facilities to load and execute Pascal programs. Aload loads by invalidating all of a process's address space, and then re-initializing from the state defined by a .RUN file.

Files: aload.pas,rundefs.pas,timedefs.pas,pathname.pas

Exported Types

```
LinkFileType = (SegFile, RunFile, DataFile, SymsFileType, QMapFileType);

Internal_Time  = record
    Weeks       : integer; Number of weeks since 17-Nov-1858
    MSecInWeek  : long;    Number of milliseconds in that week
  end;

Path_Name      = string[Path_Name_Size];  Path_Name_Size = 255

String_255     = string[255];
```

Exported Exceptions

```
exception ALoadError( s: string_255);
```

ARunLoad

Initializes a process address space.

Call:

```
procedure ARunLoad(
                RunFileName          : Path_Name;
                p                    : pointer;
                filesize             : long;
                hiskport             : port;
                LoadDebug            : boolean);
```

Parameters: *RunFileName* - run file name. This may be null is you wish to load a file that is mapped into memory.

p - optional pointer to RunFile data(or NIL)

filesize - size of run file image (in bytes) to be loaded.

hiskport - kernel port of process to be loaded.

LoadDebug - If this parameter is true, print information about the loading process.

ARunLoad initializes a process address space from *RunFileName* (or from a run file structure in memory if p is not nil), and optionally starts it executing.

ShowRun

Display contents of a .Run file.

Call:
```
procedure ShowRun(
                    p                    : pointer;
                    MapFileName          : Path_Name)
```

Parameters: *p* - optional pointer to RunFile data(or NIL)

MapFileName - file to display it on.

Writes a map for a run file into memory.

DateString

Converts an internal format date to a string.

Call:
```
function DateString(
                    date            : Internal_Time)
          : String
```

Parameters: *date* - date input as an Internal_Time

Result: A string representation of the date in the same format used by TimeUser

Use the routine T_IntToString from TimeUser instead.

LinkTypeStr

Translates a LinkFileType value to a string name.

Call:
```
function LinkTypeStr(
                              typ           : LinkFileType)
                    : string
```

Parameters: *typ* - an ennumerated type value indicating the type of a link block.

Result: a string containing the name of the link file type.

4 Bootinfo: the definition of the Boot Information Block

Implementers: Michael Jones
DavidGolub
Don Scelza

Abstract: This module provides the definition of the boot information block. This block is set up in part by MakeVMBoot when the boot file is created. The machine specific fields are filled in by the system microcode when the machine is booted. You must have physical memory privileges to access these structures.

Files: Bootinfo.pas

Exported Types

```
const
    BootBlockLocation    = #20000000000;

type ASTRecord = integer;

type
   MachineInfoRec = packed record
     case boolean of
       false: (int: integer);     { configuration comes in a word }
       true:  (
              WCSSize:    0..15;   { 0 ->  4K WCS }
                                   { 1 -> 16K WCS }
              Reserved:   0..3;
              IsPortrait: Boolean; { True  -> Portrait screen }
                                   { False -> Landscape screen }
              BoardRev:   0..31;   { IO Board revision / disk type:
                                   { 0  -> CIO board with Shugart disk
                                   { 1  -> CIO board with Micropolis disk
                                   { 16 -> EIO board
                                   {}
              OldZ80:     boolean; { True  -> Old Z80 protocol }
                                   { False -> New Z80 protocol }
              CMUNet:     boolean; { True for CMU network environment. }
                                   { False for standalone 10MBit net. }
              Reserved2:  0..3
              )
     end;

   { Boot Information Record }

   BIRecord = packed record
     case integer of
       1: ( IntBlk: array [0..255] of integer );
       2: (
            OvlTable :  array [0..11] of VirtualAddress;  { Overlays }
            VP :          VirtualAddress;   { Address of VP table }
            PV :          VirtualAddress;   {  " of PV table }
            PVList :      VirtualAddress;   {  " of PV list }
```

```
        Sector :      VirtualAddress;   {  " of sector headers }
        PCB :         VirtualAddress;   {  " of PCB Handles }
        AST :         VirtualAddress;   {  " of AST }
        AccentQueue:  VirtualAddress;   {  " of Queue headers }
        AccentFont :  VirtualAddress;   {  " of Font }
        AccentCursor :VirtualAddress;   {  " of Cursor }
        AccentScreen :VirtualAddress;   {  " of Screen segment }
        ScreenSize   : integer;
        FreeVP       :integer;          { Initial FreeVP }
        FreeAST      :integer;
        SchedProc : integer; { High level scheduling process }
        InitProc :  integer; { Initial process }
        BootChar :  integer; { Character used in boot }
        NumProc :   integer; { Number of processes set up }
        StackSize : integer; { Size of sup. stack in pages}
        GlobalSize :integer; { Size of sup. global area in pages}
        NumSVReg :  integer; { Number of regs for SVCall }
                     {** NumSVRegs is obsolete - GGR 11/16/81 **}
        TrapCode:   integer;
        TrapArgs:   VirtualAddress;
        MemBoard:   integer; { number of K of memory }
        AccentStdCursor: VirtualAddress;
        AccentRoTemp:    VirtualAddress;
        DefaultPartitionName: String[19];
        IgnoreRunFile:   Boolean;
        MachineInfo:     MachineInfoRec;
        Filler:     array [0..49-WordSize(AstRecord)] of integer;
        FirstAst:   ASTRecord;
        EtherIOArea: VirtualAddress;
        UserPtr:    VirtualAddress;     { Start of user process }
        SVContext:  record
                       SV_CS:         integer;
                       SV_GP:         integer;
                       SV_LP:         integer;
                       SV_LocalSize:  integer;
                       SV_TrapCount:  integer;
                       SV_FirstRN:    integer;
                       SV_PC_Vector: array [0..121] of integer
                    end
    )
end;

ptrBIRecord = ↑BIRecord;
```

5 CLoad: a loader for 'C' run files

Implementer: Doug Phillips

Abstract: This module provides the facilities that are used to load a process with a C program. These procedures are used by the shell and are not of general use to other programs.

File: CLoad.pas, Cfiledefs.pas

Exported Types

```
const
  CLoadNotCFile = -1;   { shouldn't conflict with any other General Return }
type

  LString       = String[ 255];

  pFirstBlock   = ↑FirstBlock;
  FirstBlock =
    record
      FileVersion       : integer;     { major version # }
      FileSize          : long;        { size of this file in bytes } {C}
      FileTimeStamp     : Internal_Time;  {C}
      SymbolAreaSize    : long;        { in pages }
      TextAreaSize      : long;        { in pages }
      DataAreaSize      : long;        { in pages } {C}
      BSSAreaSize       : long;        { in pages } {C}
      DestAddr          : long;        { word address of file in process }
      StartAddr         : long;        { of program (i.e. 'crt0') }
      MainAddr          : long;        { of program (i.e. 'main') }
      InitialLocalSize  : long;  {words needed for first stack frame locals}
      StackBaseAddress  : long;        { word address }
      StackSizeInPages  : long;
    end;

  {$IFC (WORDSIZE(FirstBlock) > 256) THEN}
    ?Error: First block is too big!
  {$ENDC}

const

  CFileVersion    = -4;
  {}
  { CFileVersion is the major version number for 'C'.  It should
  { not conflict with the version number in 'Pascal' run files.
  {}

  {}
  { Symbol entry codes
  {}
  PrimaryDef            = 10;
          {}
```

```
        { Symbol Kinds (kind of (primary) entry codes)
        {}
        PascalProcedure       = 1;  { UnUsed }
        LocalProcedure        = 2;
        GlobalProcedure       = 3;
        InitializedSymbol     = 5;
        LocalLabel            = 6;
        UndefinedSymbol       = 7;
                  { synonyms }
        UndefinedGlobal       = UndefinedSymbol;
        DefinedGlobal         = InitializedSymbol;
        DefinedLocal          = LocalLabel;
OffsetDef             = 11;
ChainHead             = 12;
AbsoluteDef           = 13;
AbsoluteLocalDef      = 14;
LibraryDef            = 90;

{}
{ Area Designators
{}
TextState             = 0;
DataState             = 1;
Data2State            = 2;  { Internal to asm }
        { synonyms }
PCInText              = TextState;
PCInData              = DataState;
PCInData2             = Data2State;

{}
{ Misc. stuff
{}
OFFSETBASE            = 16000;  { Maximum computed offset value. }
```

CLoadProcess

This procedure is used to load a process with a C program

Call:

```
Function CLoadProcess( FileName : APath_Name;
                       var FileInMem: pointer;
                       var FileSize : long;
                           Proc     : Port;
                           LoadDebug: Boolean): GeneralReturn;
```

Parameters: *FileName* - name of the C run file to be loaded. This may be null if you with to load the process with a file that is mapped into memory

FileInMem - a pointer to a run file structure that is in memory. To load a Pascal program from a file, this parameter must be nil.

FileSize - The size of the file (in bytes) to be loaded.

Proc - The kernel port of the process to be loaded.

LoadDebug - if True, print information about the loading process.

Completion Code:

Success - file was loaded - process is set up and ready to run. The file image has been freed.

CLoadNotFile - file was not a C file. FileInMem points to a copy of the file, and FileSize contains the size of the copy in bytes. (The memory image can then be passed to the Pascal loader.)

other - the process was not loaded. All memory in the parent process has been freed.

6 Clock: A 60-hz clock

Implementer: David Golub

Abstract: Provides a quick 60-hz clock routine.

File: clock.pas

IOGetTime

Gets 60-hz clock value, for relative timing.

Call:
```
function IOGetTime: long;
```

Result: Clock value (number of screen refresh cycles since system was started).

7 CommandParse: Command line parsing

Implementers: Don Scelza
Brad Myers
Michael B. Jones
David Golub
John B. Brodie

Abstract:

The modules CommandParse and ExtraCmdParse provide routines intended to ease the task of developing utilities which conform to the "standard" command syntax conventions. For historical reasons, these parsing routines are divided somewhat arbitrarily between the two modules. Start by reading this section first and it will refer you to routines in ExtraCmdParse when they are relevant.

Command parsing occurs in four distinct phases:

- The Shell transforms the user's input command into a list of words (Tokenization/partial Syntax Analysis).
- The ParseCommand routine of CommandParse further processes this word list into lists of inputs, outputs, and switches (final Syntax Analysis).
- The utility scans the three lists to ensure correctness of the parsed information (Semantic Analysis).
- The utility executes the command (Execution). In the case that the utility wants to prompt for user's input, then the first two of these phases (Tokenization and Syntax Analysis) are performed by the other parsing routines provided by CommandParse and ExtraCommandParse.

The utility should normally look at the switches first. It is especially important that the HELP switch (a switch which every standard utility is required to recognize) is looked for first. The suggested response to the HELP switch is to display the HELP'ful message; discard everything else on the command line, and then either (1) return to the shell; or (2) prompt the user for an input and then process it; or, (3) prompt the user for another command as you normally would.

Definitions:

Command Line: A line of text usually typed in by the user that invokes a command. Normally the first word is the name of the command to be invoked, followed by switches, and/or input arguments and/or output arguments.

Command File: A text file containing one or more command lines. If the user

invokes a command with the argument @*filename*, *filename* is assumed to be a command file. Command files may be nested by including the line @*nextfile* in a command file.

Character_Pool: An unbounded character string. This is the type of a raw command line.

CommandBlock: The result of the first level of parsing a command line. This parsing is normally done by the Shell. The command line is divided into words by parsing for 'white space', e.g. blanks, tabs and CRs. This is the form in which a command line is passed to a utility by the Shell via the variable **UserCommand** exported by **PascalInit**.

Command_Word_List:
The completely parsed command arguments or switches. The six parsing routines (CommandParse, ExerciseParseEngine, GetCmd, GetShellCmd, GetParsedUserInput and ParseChPool) return three of these structures, one for switches, one for input arguments and one for output arguments.

Search_Table: A private type that contains a list of words against which input words are to be matched. If an input word matches or is an unique abbreviation of one of the search table words, UniqueWordIndex will return its index. Common uses of a search table are to identify switches or subcomands.

Organization:

The routines in CommandParse do not directly interact with the user. Rather, they process a "pool" of characters which has been read-in by some other means (either by the Shell or by the routines in ExtraCmdParse). The routines in CommandParse are divided into four functional groups:

- Those which deal with nested command files.
- Those which perform the actual parsing.
- Those which deal with identifying words.
- Other miscellaneous routines that deal with character pools.

The routines in ExtraCmdParse do directly interact with the user via the keyboard/display and then return the parsed data structures representing that input.

Command File Routines:
The command file routines provide support for a LIFO stack of nested command files. This stack is maintained by these routines as a singly-linked list of file descriptors,

Command_File_List. This list always ends in a node associated with the current default input file to act as a backstop.

Parsing Routines: The parsing routines in CommandParse, parse a single line. There is one routine to parse the standard command line handed to the utility at its invokation, and another to read a line from the console and parse it. The routines to parse command files are in ExtraCommandParse.

Word Identification Routines:

Four routines are provided to support the identification of word-prefixes. A **Word_Search_Table** is the underlying data structure for these identification routines. A **Word_Search_Table** is an array of pointers to linked lists of words to be identified. Each linked list contains those words with the same first character. Thus the common case of classifying a word whose first character is sufficient for uniqueness is fast.

The initialization of such a table is fairly expensive, but searches are very fast. Thus a utility should delay the initialization of such a table until it realizes that it is actually needs to do a search.

Miscellaneous Pool Manipulators: Routines are provided to translate a PERQ character string to/from an unbounded pool of characters as used by the parsing routines.A routine is also provided to deallocate the memory of an unbounded character pool.

Constants: CommandParse also exports definitions of the separator characters of the "standard" syntax so that applications may provide their own recognizers utilizing these characters if they so desire. Note that the identifiers beginning 'shell_' represent characters which are only recognized by the Shell. They are defined herein for completeness and so that all are aware of which characters may have special meanings in the syntax.

CommandParse also exports two characters to be used in User prompts. The 'CmdChar' should be used as the last character of user prompts caused from outside a nested command file. The 'CmdFileChar' should be used inside any nested command file. The routines 'GetCmd' and 'GetParsedUserInput' in ExtraCmdParse deal with these characters automatically; while 'GetCharacterPool' does not.

File: commandparse.pas

Exported Types

```
Const                                    (* delimiters: *)
  eofChar                  = chr(0);   (* "end of file" -
                                              may NEVER appear in a word *)
  eolnChar                 = chr(#12);(* end of line *)
  single_char_quote        = '\';      (* quote just the next char *)
  quoted_text_bracket_char = '''';     (* quote an entire string *)
  env_var_bracket_char     = '↑';      (* substitute from the environment *)
  env_quoted_bracket_char  = '"';      (* environ sub into a quoted string *)

  switch_leadin_char       = '-';
  value_marker_char        = '=';
  command_file_leadin_char = ';
  comment_leadin_char      = '#';
  in_out_separator_char    = '~';

  shell_special_args_start = '[';      (* these are recognized by *)
  shell_special_args_stop  = ']';      (* the shell only *)
  shell_input_redirect     = '<';
  shell_output_redirect    = '>';      (* parsing routines herein attach *)
  shell_sequential_command = ';';      (* no special meaning to them and *)
  shell_piped_command      = '|';      (* treat them as if they were *)
  shell_parallel_command   = '&';      (* alpha-numerics *)

                             (* characters to be appended to prompts *)
   CmdChar                  = Chr(#200+24);  (* use outside command files *)
   CmdFileChar              = Chr(#200+26);  (* use inside command files *)

   CommandParseVersion      = '5.7 of 3 Jun 84';
   MaxCmndString            = 255;

(* word identification routines *)
Const                          (* errors returned by UniqueWordIndex *)
   WS_NotFound    = -1;           (* word-prefix not found *)
   WS_NotUnique   = -2;           (* word-prefix not unique *)


Type
   pCmnd_String = ↑ Cmnd_String;
   Cmnd_String = String[MaxCmndString];

   pWord_String = ↑ Word_String;
   Word_String = String[1];  (*DO NOT try to STORE into a Word_String*)

   pCommand_File_List = ↑Command_File_List;
   Command_File_List = RECORD          (* the command file stack *)
         cmdFile: Text;
         isCharDevice: Boolean;
         next: pCommand_File_List;
      END;

   Word_Type = (in_arg, out_arg, switch_arg, switch_value, command_file);

   pCommand_Word_List = ↑ Command_Word_List;
```

```
(* Use AllocCommandNode (see below) to make one of these *)

Command_Word_List = packed record  (*the structure of a parsed command*)
      ptrWordString: pWord_String;
      DeallocWordString: boolean;   (* DO NOT MESS WITH THIS FIELD *)
      case WordClass: Word_Type of
      in_arg,
     out_arg,
command_file:(NextArg:          pCommand_Word_List);
  switch_arg:(NextSwitch:       pCommand_Word_List;
              ValueOfSwitch:    pCommand_Word_List;
              CorrespondingArg: pCommand_Word_List);
switch_value:((*nothing*));
   end;

pWord_Search_Table = POINTER; (*structure of table is strictly private*)
```

The following types are actually exported by CommandDefs but are included here for the convenience of the reader

```
Character_Pool  = packed array [0..0] of char;
                    {* an unbounded chunk of characters *}

pCharacter_Pool = ↑Character_Pool;

Char_Pool_Index = long;

CommandBlock    = record
      WordCount      : long;            { number of words }
      WordDirIndex   : Char_Pool_Index; { Byte index to word dictionary }
      WordArrayPtr   : pCharacter_Pool;
      WordArray_Cnt  : Char_Pool_Index;
   end;
```

Command File Routines

InitCmdFile

Initializes the first node of* inF *to be a valid Text File corresponding to the keyboard. This must be called before any other command file routines.

Call: `procedure InitCmdFile(var InF: pCommand_File_List);`

Parameters: *InF* - Output variable which will point to the top of the stack of command files.

Initializes the first node of **inF** to be a valid Text File corresponding to the keyboard. This must be called before any other command file routines. The application should then read from inF↑.cmdFile. E.g. ReadLn(inFile↑.cmdFile, s); or while not eof(inFile↑.cmdFile) do ... Use popup only if inF↑.next = NIL (means no cmd File). The CmFile is a fileSystem file if inF↑.IsCharDevice is false. InF will never be NIL. The user should not modify the pCommand_File_List pointers; use the procedures provided.

OpenCmdFile

Opens a new command file and pushes it onto the specified stack of open command files.

Call:

```
function  OpenCmdFile(FileName:       pWord_String;
                      var InF:    pCommand_File_List)
          : GeneralReturn
```

Parameters: *FileName* - the name of the command file to be opened. The application is responsible for ensuring that the file appears in a correct context.

InF - the list of command files. This was originally created by InitCmdFile and is maintained by these routines. If FileName is a valid file, a new entry is put on the front of InF describing it. If there is an error, then InF is not changed. In any case, InF will always be valid.

Result: Returns success if the new command file is opened successfully and failure otherwise.

This function will prepare a command file for use by the standard pascal I/O routines. The user should give OpenCmdFile the filename as parsed by one of the parsing routines contained in this CommandParse module. The application is expected to ensure that the command file has appeared in a correct context for that application. These checks might include ensuring that no other words appeared within the input line containing the command file. This function maintains a stack of command files so that command files may contain other command files. Be sure to call InitCmdFile before calling this procedure.

ExitCmdFile

Removes the top command file from the list.

Call:

```
procedure ExitCmdFile(var inF: pCommand_File_List)
```

Parameters: *InF* - the list of command files. It must never be NIL. The top entry is removed; except when attempting to remove last command file, when it is simply re-initialized to be the backstop default input file. It is OK to call this routine even when at the last entry of the list.

Result: One element is popped off of the command stack, InF.

Call this whenever the end of a command file is reached.

ExitAllCmdFiles

Removes all command files from the given list.

Call:

```
procedure ExitAllCmdFiles(var InF: pCommand_File_List)
```

Parameters: *InF* - the list of command files. It must never be NIL. All entries but the last are removed.

Result: InF is reset to a single list node attached to the backstop default input file.

Use when a fatal error has been found or upon reciept of either a SigLevel2Abort or a SigLevel3Abort in order to reset all command files.

DstryCmdFiles

Removes all command files from the given list. All entries are removed and InF is set to NIL. InitCmdFile must be called again before any command file stack routines may be used.

Call:
```
procedure DstryCmdFiles(var InF: pCommand_File_List)
```

Parameters: *InF* - the list of command files to be released.

Result: InF is set to NIL.

Parsing Routines

InitCommandParse

Initializes the parser by marking the parsing state tables as un-initialized. The Parser, when first invoked, will notice that the parsing table need to be initialized and will do so. Thus delaying a possibly lengthy initialization process until the data is actually needed.

Call:
```
Procedure InitCommandParse
```

Result: Resets the private global table initialized flag to FALSE.

DestroyCommandParse

Deallocates the parsing DFA table.

Call:
```
Procedure DestroyCommandParse
```

Result: Resets the private global table inited flag to FALSE.

ParseCommand

Transforms the word list passed to the program into the parsed data lists of input, outputs, and switches.

Call:
```
Function ParseCommand(var inputs: pCommand_Word_List;
                      var outputs: pCommand_Word_List;
                      var switches: pCommand_Word_List)
                    : GeneralReturn
```

Parameters: *inputs* - list to contain the input words.

outputs - list to contain the output words.

switches - list to contain the switches.

Result: Sets the three lists and returns a GeneralReturn code indicative of the parse results.

This routine parses **UserCommand** which is the command line that is passed to the utility from the shell via an export in **PascalInit**. The program name is normally passed to the program as the first word in the list. The word is stripped and ignored by ParseCommand.

ParseChPool

Parses the given unbounded pool of characters.

Call:

```
Function ParseChPool(ChPool: pCharacter_Pool;
                         PoolLength:  Char_Pool_Index;
                     var inputs:  pCommand_Word_List;
                     var outputs: pCommand_Word_List;
                     var switches: pCommand_Word_List)
            : GeneralReturn
```

Parameters: *ChPool* - pointer to the beginning of the pool to be parsed.

PoolLength - number of characters in the pool.

inputs - list to contain the input words.

outputs - list to contain the output words.

switches - list to contain the switches.

Result: Sets the three lists and returns a GeneralReturn code indicative of the parse results.

This routine may be called if the utility has read in a new command line from the user and wants it to be parsed by the standard parser.

ExerciseParseEngine

This routine is the Parser. It is called by all other routines which perform parsing. It scans the given character pool (augmented if necessary by reading another pool) and constructs the parsed data lists of inputs, outputs, and switches.

Call:

```
Function  ExerciseParseEngine
                              (ChPool: pCharacter_Pool;
                                 PoolLength: Char_Pool_Index;
                        procedure ReadPool(var Pool: pCharacter_Pool;
                                           var PLen: Char_Pool_Index);
                                  var inputs: pCommand_Word_List;
                                  var outputs: pCommand_Word_List;
                                  var switches: pCommand_Word_List)
            : GeneralReturn;
```

Parameters: *ChPool* - the first hunk of characters to be parsed.

PoolLength - The length of the first pool of characters. This parameter may be zero; in which case NIL parse structures will be returned.

ReadPool - a procedure to read successive pools of characters, if required by the parsing actions.

inputs - the list to contain the words recoginzed as inputs.

outputs - the list to contain the words recognized as outputs.

switches - the list to contain the words recognized as switches.

Result: Places the parsed lists into the parameters and returns a GeneralReturn code of either Success if all went well or an error code indicative of the error.

Assumes that InitCommandParse has been called. This routine is called by a utility that has read (or wants ExerciseParseEngine to read) an input line from the console. If the next line may be coming from a command file, call GetParsedUserInput instead. ExerciseParseEngine should be called vigorously for at least 20 minutes twice weekly in order for the software to remain in top condition.

AllocCommandNode

Allocates a new node to be inserted into the parsed data structures. User is responsible for establishing the correct linkages. This routine merely creates a node with the correct string text value for the word.

Call:
```
Function AllocCommandNode(WordClass :  Word_Type;
                          WordString:  Cmnd_String)
                    : pCommand_Word_List
```

Parameters: *WordClass* - the class desired for the new node.

WordString - the text of the new word.

Result: returns a pointer to the newly allocated node.

DestroyCommandList

Deallocates a parsed data structure.

Call:
```
Procedure DestroyCommandList(var argList: pCommand_Word_List)
```

Parameters: *argList* - a pointer to the structure to be released.

Result: argList is set to NIL.

AlwaysEof

Support routine for callers of ExerciseParseEngine. This routine sets up a character pool which contains an end-of-file marker. This will signal the parsing activity to stop.

Call:
```
Procedure AlwaysEof(var ChPool:          pCharacter_Pool;
                    var PoolLength:      Char_Pool_Index)
```

Parameters: *ChPool* - pointer to the pool to contain the end-of-file marker.

PoolLength - length of the eof buffer (always set to 1 by this routine).

Routines for indentifying word values

InitWordSearchTable

Creates an empty search table for use in word identification.

Call:
```
procedure StdError(var table:      pWord_Search_Table;
                       CaseSensitive:       boolean)
```

Parameters: *table* - will be set to the address of the new search table.

CaseSensitive - TRUE if the words in this table should retain their capitalization during the identification processing.

Result: The address of a new search table is returned in 'table'.

This routine must be called before any of the other search table routines.

AddSearchWord

Merges the given word into the search table under the given key.

Call:
```
Procedure AddSearchWord(table: pWord_Search_Table;
                        WordKey: integer;
                        WordString: Cmnd_String)
```

Parameters: *table* - POINTER to a search table.

WordKey - identification of the word. The key may be any arbitrary non-negative integer. The negative numbers are reserved by the identification and parsing routines to indicate errors.

WordString - the text of the word.

Exceptions: *Impossible* - raised if table is Nil, or WordKey is negative, or WordString is zero length

Table must be the result of an InitWordSearchTable call. Adds a word to the search table.

DeleteSearchWord

Expunges the given word from the search table.

Call:
```
Procedure DeleteSearchWord(table:      pWord_Search_table;
                           WordString:       Cmnd_String)
```

Parameters: *table* - POINTER to a search table

WordString - the text of the word to be deleted.

Exceptions: *Impossible* - raised if table is Nil, or WordString is zero length

Table must be the result of an InitWordSearchTable call.

DestroySearchTable

Expunges the data for the given search table.

Call:
```
Procedure DestroySearchTable(var table:  pWord_Search_Table)
```

Parameters: *table* - POINTER to the search table to be destroyed.

Result: table is set to NIL.

DestroySearchTable deallocates the memory of a search table.

UniqueWordIndex

Locates the given word in the given search table. Returns both the key and the text of the word found.

Call:
```
Function UniqueWordIndex(table: pWord_Search_Table;
                         ptrWordString:      pWord_String;
                     var WordText:     Cmnd_String)
        :integer
```

Parameters: *table* - the search table in which to attempt the identification.

ptrWordString - a pointer to the text of the word-prefix to be found.

WordText - the text of the found word.

Result: Returns the word-key and the entire text of the identified word if and only if the given word-prefix is in the table is not found in the table or a WS_NotUnique if the word-prefix is not unique. In either of the latter error conditions WordText will contain the erroneous word-prefix.

Assumes that table has been initialized via InitWordSearchTable and AddSearchWord.

Miscellaneous conversion routines

ConvertPoolToString

Transforms an arbitrary portion of the given unbounded pool of characters into a PERQ Pascal String.

Call:
```
procedure ConvertPoolToString(ChPool: pCharacter_Pool;
                         FirstChar:      Char_Pool_Index;
                         StringLength:   Char_Pool_Index)
                : Cmnd_String
```

Parameters: *ChPool* - a pointer to the unbounded hunk of characters.

FirstChar - the beginning position (zero based) within ChPool of the desired string.

StringLength - the number of characters to be moved from the pool to the string.

Result: Returns a PERQ Pascal String.

Exceptions: *StrTooLong* - if asked to create a string longer than 255 characters.

The result of a function is to be placed in its local #0.

ConvertStringToPool

Transforms a PERQ Pascal String into an unbounded pool of characters suitable for use by the ParseChPool routine. Is symetric with ConvertPoolToString.

Call:
```
procedure ConvertStringToPool(CnvStr:    Cmnd_String;
                              var ChPool:    pCharacter_Pool;
                              var PoolLength:   Char_Pool_Index)
```

Parameters: *CnvStr* - the PERQ Pascal String to be converted.

ChPool - set to the address of the created pool.

PoolLength - the number of characters in the resultant pool.

Result: Sets the parameters 'ChPool' and 'PoolLength' to describe the new unbounded hunk of characters.

DestroyChPool

Deallocates storage of a given unbounded pool of characters.

Call:
```
procedure DestroyChPool(var ChPool: pCharacter_Pool;
                        var PoolLength:  Char_Pool_Index)
```

Parameters: *ChPool* - pointer to the pool to be released; is set to NIL.

PoolLength - number of characters to be released; is zeroed.

Result: Both ChPool and PoolLength are modified to describe an empty pool.

WordifyPool

Transforms the given unbounded pool of characters into a word list suitable for passing to a Spawn'd child process.

Call:
```
Function WordifyPool(ChPool: pCharacter_Pool;
                     PoolLength: Char_Pool_Index;
                 var        WordStruct:        CommandBlock):

GeneralReturn
```

Parameters: *ChPool* - pointer to the beginning of the pool to be 'wordified'.

PoolLength - number of characters in the pool.

WordStruct - a buffer to contain the word list description.

Result: Sets the WordStruct and returns a GeneralReturn code indicative of the wordification results.

GetIthWordPtr

Retrieves a pointer to the desired word from the given word list.

Call:
```
procedure GetIthWordPtr(i:   long;
                        CmndBlock: CommandBlock)
                  :  pWord_String
```

Parameters: *i* - the number (one-based) of the desired word.

CmndBlock - the block from which to fetch the word.

Result: Returns a pointer to the requested word or NIL if the desired number is out of bounds.

8 CommandDefs: Definitions for the command structure passed between Accent programs.

Implementers: John Brodie
David Golub

Abstract: Definitions for the command structure passed between Accent programs.

Files: commanddefs.pas,commandparse.pas

Exported Types

```
{ Error General Return Values  }

const
        CmdParse_Error_Base  =  4200;

        ErBadSwitch          = CmdParse_Error_Base  +  1;
        ErBadCmd             = CmdParse_Error_Base  +  2;
        ErNoSwParam          = CmdParse_Error_Base  +  3;
        ErNoCmdParam         = CmdParse_Error_Base  +  4;
        ErSwParam            = CmdParse_Error_Base  +  5;
        ErCmdParam           = CmdParse_Error_Base  +  6;
        ErSwNotUnique        = CmdParse_Error_Base  +  7;
        ErCmdNotUnique       = CmdParse_Error_Base  +  8;
        ErNoOutFile          = CmdParse_Error_Base  +  9;
        ErOneInput           = CmdParse_Error_Base  +  10;
        ErOneOutput          = CmdParse_Error_Base  +  11;
        ErIllCharAfter       = CmdParse_Error_Base  +  12;
        ErBadQuote           = CmdParse_Error_Base  +  13;
        ErAnyError           = CmdParse_Error_Base  +  14;

   ParseInternalFault          = CmdParse_Error_Base  +  15;
   ParseWordTooLong            = CmdParse_Error_Base  +  16;
   ParseIllegalCharInSwName    = CmdParse_Error_Base  +  17;
   ParseIllegalCharInSwVal     = CmdParse_Error_Base  +  18;
   ParseIllegalCharInEnvNam    = CmdParse_Error_Base  +  19;
   ParseIllegalCharInQuoted    = CmdParse_Error_Base  +  20;
   ParseOnlyCmdAllowed         = CmdParse_Error_Base  +  21;
   ParseIllegalCharInInRed     = CmdParse_Error_Base  +  22;
   ParseIllegalCharInOutRed    = CmdParse_Error_Base  +  23;
   ParseIllegalCharInShPara    = CmdParse_Error_Base  +  24;

type
   Character_Pool  = packed array [0..0] of char;
                       {* an unbounded chunk of characters *}

   pCharacter_Pool = ↑Character_Pool;

   Char_Pool_Index = long;

   CommandBlock    = record
       WordCount    : long;            { number of words }
```

```
    WordDirIndex : Char_Pool_Index; { Byte index to word dictionary}
      WordArrayPtr   : pCharacter_Pool;
      WordArray_Cnt  : Char_Pool_Index;
  end;
```

Null-CommandBlock

Null_CommandBlock is used to get a new, empty CommandBlock

Call: `function Null_CommandBlock: CommandBlock`

Result: A new empty CommandBlock

9 Configuration: provides Perq configuration information

Implementers: David Golub

Abstract: Configuration is used to provide information about the hardware configuration of the current machine. To use this module a process must have access to physical memory.

Files: Configuration.pas

Exported Types

```
type
    Cf_IOBoardType = (Cf_CIO,       { Perq1 }
                      Cf_EIO);      { Perq2 }

    Cf_MonitorType = (Cf_Landscape,
                      Cf_Portrait);

    Cf_NetworkType = (Cf_CMUNet,
                      Cf_10MBitNet);
```

Exported Exceptions

```
exception ConfigurationError( s: string_255);
```

CF-IOBoard

This procedure is used to obtain the type of the I/O board.

Call: `function CF_IOBoard : CF_IOBoardType`

Result: Returns CF_CIO if this is a PERQ I/O board, CF_EIO if it is a PERQ2

CF-Monitor

This procedure is used to obtain the type of display that is on the machine

Call: `function CF_Monitor: CF_MonitorType`

Result: Return CF_CIO if this is a PERQ I/O board, CF_EIO if it is a PERQ2.

CF-OldZ80

This procedure is used to determine the protocol that is to be used to communicate with the I/O board Z80. Accent no longer supports the "old" Z80 protocols.

Call: `function CF_OldZ80 : boolean`

Result: Returns true if the protocol is old Z80 - This procedure should always return false.

CF-Network

Provides information about the type of network that is in use.

Call: `function CF_Network: CF_NetworkType`

Result: Return CF_10MBitNet is there is a standard Ethernet or if there is no network. If the machine is running in the Carnegie -Mellon University network environment return CF_CMUNet.

10 Dynamic: Pascal dynamic allocation routines

Implementer: W.J. Hansen

Abstract: Implements Pascal dynamic allocation - New and Dispose. Also provides routines CreateHeap, ResetHeap and DestroyHeap to make, empty, and get rid of heaps. In general calls to these routines should only be generated by the compiler.

File: dynamic.pas

Design: InitDynamic must be called before calling any other routine in this module. Normally it is called by PascalInit before the user's main program gets control.

Dynamic maintains chains of heaps. Each element is (HeapLimit + 1) words long, but the Last PostFixSize words are occupied by control information. Generally, HeapLimit is 32767 so all links in a heap are positive integers.

Memory of a given size with a given alignment may be allocated from any heap. If the heap is full (doesn't contain enough free memory to meet the request), then an additional heap is attached to the current one. These additional heaps are kept in the chain on the NextHeap field.

Memory cells may be deallocated by Dispose. Free memory within each heap is linked into a circular freelist in order of address. Each free node is at least two words long and is of the form:

```
record
  Next:   Integer;        index of next free block
  Len:    Integer;        length of this free block
  Tag:    Integer;        set to a known value in free nodes
  Rest:   array [1..Length - 2] of integer;
end;
```

Requests larger than HeapLimit-PostFixSize are met by doing a direct ValidateMemory or InvalidateMemory.

Exported Types

```
HeapNumber = integer;
```

Exported Exceptions

exception NilPointer;

NilPointer is raised when a Nil pointer is used or passed to Dispose.

exception BadPointer;

BadPointer is raised when the pointer passed to dispose cannot really be a pointer to a node to be freed.

exception TooManyHeaps;

TooManyHeaps is raised when CreateHeap is called and there are already MaxHeaps allocated. Fatal even if handled (raises ExitProgram when returned to).

```
exception AccentError(R: GeneralReturn; M: String);
```
Accent did not return success from some system call.

```
Parameters:
      R - The return value from Accent.
      M - A message describing what the caller was doing.
```

```
exception BadHeap( H: HeapNumber );
```
BadHeap is raised when the heap passed to New is invalid. This may mean that the heap has been improperly modified or that the pointer itself is improper or uninitialized. (It should be initialized with CreateHeap.)

```
Parameters:
      H - Pointer that caused the error.
```

```
exception BadAlignment;
```
Raised if the alignment to NewP is not a power of two.

InitDynamic

Initializes Pascal dynamic memory allocation

Call: `procedure InitDynamic`

InitDynamic must be called only once and before any call to any other routine in this module.

CreateHeap

Validate memory for a heap and initialize its free list.

Call: `function CreateHeap : HeapNumber`

Result: HeapNumber of the new heap.

Exceptions: *TooManyHeaps* - If the process is already using MaxHeaps number of heaps

ResetHeap

Return all storage in heap S to free list.

Call: `procedure ResetHeap(S : HeapNumber)`

Parameters: *s* - HeapNumber of the heap that is to be reinitialized.

DestroyHeap

Release storage for heap S

Call: `procedure DestroyHeap(S : Heap Number)`

Parameters: *s* - HeapNumber of heap to destroy.

DisposeP

Deallocate memory

Call:

```
procedure DisposeP(
            var Where          : pointer;
                Len            : integer)
```

Parameters: *Where* - Pointer to the record to release.

Len - Length in words.

Exceptions: *BadHeap* - if the record is in some heap that does not have reasonable values in the control fields.

BadPointer - if *Where* + *Length* extends off the end of heap, or if the node to be Disposed overlaps some node that is already free.

NilPointer

Normally the node pointed to by *where* is added to the free list of the heap in which it was allocated. If the length of the node is greater than MaxAvail, then the cell is released with InvalidateMemory. Otherwise it is assumed to be part of a heap starting at the next lower multiple of HeapLimit + 1.

NewP

Alloacate Memory

Call:

```
procedure NewP(
                  S                : HeapNumber;
                  A                : integer;
            var   Where            : pointer;
                  L                : integer )
```

Parameters: *s* - Number of heap in which to allocate. 0 means the default data heap.

A - Alignment of node in words relative to beginning of segment. Must be a power of 2. 0 represents 2**16, the maximum value.

Where - Set to point to the memory that was allocated. If the data segment is full and cannot be increased, P is set to nil.

L - Length in words. 0 represents 2**16, the maximum. Negative values are interpreted as (L + 2**16).

Exceptions: *FullMemory* - if NewP tries to expand the segment, but there is not enough physical memory to do so.

Allocates a new node in the specified heap. If the size of the requested node is greater than the size of the heap, the storage is allocated by using the ValidateMemory kernel call.

11 Except: Exporter of exceptions

Implementers: John Strait
George Robertson
Eugene Ball

Abstract: Provides runtime support for Pascal exceptions. Exports the definitions for many of the standard exceptions.

File: except.pas

Other modules that export exceptions are: Dynamic,EtherUser,PascalInit, Pmatch,Reader, Spice_String, Stream and TimeUser.

Exported Exceptions:

```
Exception Abort( Message: String );
Exception Dump( Message: String );
```

`Exception DivZero;`	division by zero
`Exception MulOvfl;`	overflow in multiplication
`Exception StrIndx;`	string index out of range
`Exception StrLong;`	string to be assigned is too long
`Exception InxCase;`	array index or case expression out of range
`Exception UndfQcd;`	execution of an undefined Q-code
`Exception UndfInt;`	undefined device interrupt detected
`Exception MParity;`	memory parity error
`Exception EStack;`	E-stack wasn't empty at INCDDS
`Exception OvflLI;`	Overflow in conversion to integer from Long Integer
`Exception NormMsg;`	normal IPC msg pending
`Exception EmergMsg;`	emergency IPC msg pending
`Exception UndReal;`	floating point underflow
`Exception OvrReal;`	floating point overflow
`Exception RtoIOvfl;`	floating point truncate error
`Exception RealDivZero;`	floating point divide by zero
`Exception NextOp;`	NextOp across a page boundary
`Exception BadExit;`	EXITT or EXGO falling off stack

RaiseP

RaiseP is called to raise an exception.

Call: `procedure RaiseP( ES, ER, PStart, PEnd: Integer )`

Parameters: *ER* - Routine number of the exception to be raised.

ES - Segment number of the exception to be raised.

PStart - Pointer to the original parameters (as an offset from the base of the stack).

PEnd - Pointer to the first word after the original parameters (as an offset from the base of the stack).

The compiler generates a call to RaiseP in response to

raise SomeException(original parameters)

in the following way

Push original parameters onto the MStack. RAISE SegmentNumber(SomeException) RoutineNumber(SomeException) ParameterSize

The microcode calls RaiseP in the following way.

Push parameters onto the MStack if appropriate. ParameterSize : = WordsOfParameters. Error : = ErrorNumber, Goto(CallRaise).

Where CallRaise does the following.

SaveTP : = TP. Push ExcSeg onto the MStack. Push Error onto the MStack. Push SaveTP-ParameterSize + 1 onto the MStack. Push SaveTP + 1 onto the MStack. call RaiseP.

InitExceptions

InitExceptions tells the microcode what segment number to use when raising its own exceptions. The segment number is the one that the system assigns to this module.

Call: `procedure InitExceptions`

12 ExtraCmdParse: More help in parsing a command line

Implementers: Brad Myers
John B. Brodie

Abstract: The modules CommandParse and ExtraCmdParse provide routines intended to ease the task of recognizing the "standard" command syntax conventions. See the section on CommandParse for a explaination of parsing conventions.

The routines provided by ExtraCommand parse input from Command Files. If you are writing a really first class utility which wants to be able to handle input from command files as well as single line input, you should use these parsing routines along with the command file management routines in CommandParse.

ExtraCmdParse also provides routines to get user input from the console and to ask yes/no questions of the user.

Files: extracmdparse.pas, cmdparse.pas

Exported Types

```
Const          (* anomolous conditions reported by GetCmd and GetShellCmd *)
   Cmd_NotFound             = WS_NotFound;  (* first input word not
                                               found in search table *)
   Cmd_NotUnique            = WS_NotUnique; (* first input word-prefix
                                               not unique *)
   Cmd_EmptyCmdLine         = -3;           (* user line was empty *)
   Cmd_NotInsMaybeSwitches  = -4;           (* user line contained no inputs
                                               but maybe has switches *)
   Cmd_SomeError            = -5;           (* some error has occured --
                                               code is in ErrorGR *)

                                  (* indicators returned by GetConfirm *)
   Confirm_YES              = 1;
   Confirm_NO               = 2;
   Confirm_Switches         = 3;

  Prompt_Indention_String = '    ';(* amount to indent prompts for params *)
```

GetCmd

GetCmd will obtain a command input from the top file on the given file list. Note that the top file may be the console. GetCmd will completely parse the input line. This routine and GetShellCmd are designed for utilities that expect the first input word to be a subcommand specifiying an operation for the utility to perform. Thus a match for the first word is searched for in the specified SearchTable.

Call:

```
Function GetCmd   (Prompt :       Cmnd_String;
                   SearchTable:     pWord_Search_Table;
                  var CmdName:      Cmnd_String;
                  var InF:          pCommand_File_List;
                  var inputs:       pCommand_Word_List;
```

```
var outputs:      pCommand_Word_List;
var switches:     pCommand_Word_List;
var ErrorGR:      GeneralReturn): integer;
```

Parameters: *Prompt* - the prompt string to print for the user. Do not put the prompt separator (>) on the end of the prompt; GetCmd will do that for you. If reading from a command file, GetCmd changes the prompt appropriately. Prompt is always displayed on the current default output file.

SearchTable - the word search in which to attempt to locate the user's first input.

CmdName - buffer to contain the text of the user's first input.

InF - command file list on which to nest command files (if any).

inputs - list to contain the user's input words.

outputs - list to contain the user's output words.

switches - list to contain the user's switch selections.

ErrorGR - GeneralReturn code indicating status of the parse processing.

Result: Returns the UniqueWordIndex of WordKey for CmdName in SearchTable or

Cmd_NotFound	- first input word not found in SearchTable
Cmd_NotUnique	- first input word not unique
Cmd_EmptyCmdLine	- command line was empty
Cmd_NotInsMaybeSwitches	- no inputs appeared on the line -but there may be switches
Cmd_SomeError	-error was discovered (ErrorGR contains error code)

on the current default output file.

After parsing the command line, GetCmd will examine the parsed structures to determine if a command file reference was the first (and only) item in the command. If a command file is found, that file is prepended to the command file list and a null command is signalled to the caller. If no command file is found, then an attempt to locate the first input argument in the given search table is made. The results of that search are returned to the caller along with the text of the first argument, and the lists of inputs (sans first arg), outputs, and switches.

Assumes that InitCmdFile and InitCommandParse has been called. Also assumes that SearchTable has been appropriately initialized via InitWordSearchTable and AddSearchWord calls and that the current default output file has been rewritten.

GetShellCmd

This routine is similar to GetCmd except that it works on the command line specified to the Shell. It is should be used by programs that use GetCmd so that the Shell command line may be parsed in a similar manner. Command files are handled by GetShellCmd in a manner like GetCmd.

Call:

```
Function  GetShellCmd(SearchTable: pWord_Search_Table;
                      var CmdName: Cmnd_String;
                      var inF: pCommand_File_List;
                      var inputs: pCommand_Word_List;
                      var outputs: pCommand_Word_List;
                      var switches: pCommand_Word_List;
                      var ErrorGR: GeneralReturn): integer
```

Parameters:

SearchTable - the word search in which to attempt to locate the user's first input.

CmdName - buffer to contain the text of the user's first input.

InF - command file list on which to nest command files (if any).

inputs - list to contain the user's input words.

outputs - list to contain the user's output words.

switches - list to contain the user's switch selections.

ErrorGR - GeneralReturn code indicating status of the parse processing.

Result:

Identical to GetCmd. Viz: returns the UniqueWordIndex of WordKey for CmdName in SearchTable or

Cmd_NotFound	- first input word not found in SearchTable
Cmd_NotUnique	- first input word not unique
Cmd_EmptyCmdLine	- command line was empty
Cmd_NotInsMaybeSwitches	- no inputs appeared on the line - but there may be switches
Cmd_SomeError	- error was discovered (ErrorGR contains error code)

Assumes that InitCmdFile and InitCommandParse have been called and that SearchTable has been appropriately initialized via InitWordSearchTable and AddSearchWord calls.

GetShellCmd performs basically the same processing as GetCmd. Except that GetShellCmd does not prompt the user for input, but rather utilizes the word list information passed to the utility by the Shell.

A short example of GetCmd/GetShellCmd:

```
.
InitCmdFile(Input_File_List);
InitCommandParse;
```

```
.
.
(* get the first input from the shell *)

Cmd_Index := GetShellCmd(CmdTable,
                         Cmd_Name,
                         Input_File_List,
                         Inputs, Outputs, Switches,
                         GR);

(* command file nesting handled transparently to the caller *)

while The_Utility_Has_Something_To_Do do
  begin
    case Cmd_Index of
    Cmd_SomeError: GRWriteStdError(GR, GR_whatever, '');
                          (* see SaltError *)
Cmd_NotIns_MaybeSwitches: doSwitches(Switches);
   Cmd_EmptyCmdLine: (* do nothing *)
     Cmd_NotUnique: GRWriteStdError(ErCmdNotUnique, GR_whateverVerb_String);
      Cmd_NotFound: GRWriteStdError(ErBadCmd, GR_whatever, Verb_String);
 some_cmd_key: do_some_cmd(whatever_the_command_needs,.....
                 Inputs, Outputs, Switches);
          (* deal with all inputs, outputs, and switches *)
          (* as is appropriate for the given command  *)

some_other_cmd_key:
        do_some_other_cmd(whatever_that_command_needs,.....
                 Inputs, Outputs, Switches);
          (* deal with all inputs, outputs, and switches *)
          (* as is appropriate for the given command verb *)
  :
  :
  :
    end;

    (* get the next input from the user *)

    Cmd_Index := GetCmd('utility name',
                 CmdTable,
                 Cmd_Name,
                 Input_File_List,
                 Inputs, Outputs, Switches,
                 GR);

    (* command file nesting handled transparently to the caller *)

  end;
```

GetParsedUserInput

GetParsedUserInput will obtain a command input from the top file on the given command list. It will then parse that command and return to the caller the lists of inputs, outputs, and switches. This routine is intended for those applications in which the rules for parsing user input into words is desired but for which the application desires to perform all processing of the parsed words. (That is, use this if you don't want the command-file and word-identification effects of the GetCmd routine).

Call:

```
Function  GetParsedUserInput(prompt: Cmnd_String;
                             var inF: pCommand_File_List;
                             var inputs: pCommand_Word_List;
                             var outputs: pCommand_Word_List;
                             var switches: pCommand_Word_List)
                : GeneralReturn
```

Parameters: *Prompt* - the prompt string to print for the user. Prompt string is displayed as is, no modifications made. Prompt is always displayed on the current default output file.

InF - command file list from which to read the user's command.

inputs - list to contain the user's input words.

outputs - list to contain the user's output words.

switches - list to contain the user's switch selections.

Result: Returns a GeneralReturn code indicating either Success, if all went well, or an appropriate error code.

Assumes that InitCmdFile and InitCommandParse have been called and that the current default output file has been rewritten.

GetParsedUserInput is a general routine used to prompt the user for input and then return parsed data to the caller. It makes no assumptions about the meaning of the words it reads (other than the normal syntactic rules). Thus it does not automatically nest command files nor does it attempt to identify the first input word.

GetConfirm

Handles a question that is to be answered Yes or No where the answer should come from the keyboard. Prompt followed by default (if any) is printed. Prompt may be null. If illegal input is typed, GetConfirm re-asks but doesn't use prompt.

Call:

```
Function GetConfirm (prompt :  Cmnd_String;
                       def :    integer;
                     var switches: pCommand_Word_List)
                  : integer
```

Parameters: *Prompt* - the prompt to display for question. Prompt is always displayed on the current default output file.

Def - Index of the default answer: Confirm_YES = true or yes; Confirm_NO = false or no; other numbers mean no default.

Switches - set to NIL or a list of switches specified. Be sure to handle the switches first since one might be HELP.

Result: Confirm_YES if true or yes.

Confirm_NO if false or no.

Confirm_Switches if naked return when no default and switches <> NIL. This means that there was no argument but a switch was entered. If an answer is still needed, the application should re-call GetConfirm.

Assumes that InitCmdFile and InitCommandParse have been called and that the current default output file has been rewritten.

The following semantic rule are enforced for the user's answer;

1) no outputs allowed (e.g. ~ is illegal)
2) no command files allowed (e.g. is illegal)
3) only zero or one inputs allowed.
4) if zero inputs then either switch(es) must exist or the line must be empty.
5) if one input then NO switch(es) must exist and it must be either Yes or No.

GetCharacterPool

GetCharacterPool should be used by those applications which wish to perform their own command input line parsing. This routine will interact with the user to obtain an unbounded raw pool of characters.

Call:

```
Procedure GetCharacterPool(prompt: Cmnd_String;
                           var InputFile: Text;
                           var ChPool: pCharacter_Pool;
                           var PoolLength: Char_Pool_Index)
```

Parameters: *Prompt* - the prompt string to print for the user. GetCharacterPool displays this string exactly as given; no changes made. Prompt is always displayed on the current default output file.

InputFile - the file from which to read the pool of chars. If this is set to **input** the characters will be read from the console.

ChPool - a pointer to the pool read.

PoolLength - number of characters read.

Result: All results returned via the parameters.

Assumes that the current default output file has been rewritten.

GetCharacterPool is used to obtain from the user a raw hunk of characters. GetCharacterPool makes no assumptions about the meaning of the characters nor does it attempt any processing of the characters.

Note that since GetCharacterPool attempts no interpretation of the characters it reads, it will not recognize quoted end-of-line characters and therefore it will not read continuation lines.

13 IPCRecordIO: Simple routines to send and receive messages

Implementer: Richard F. Rashid

Abstract: Routines for sending and receiving simple Pascal records.

File: ipcrecordio.pas

SendRecord

Send a Pascal record (which does not contain a port reference) to a port.

Call:

```
function SendRecord(
                localport               : Port;
                remoteport              : Port;
                id                      : long;
                MsgType                 : long;
                recptr                  : Pointer;
                recsize                 : integer)
        : GeneralReturn;
```

Parameters: *localport* - a port in the current process (usually used for a reply to the sent message, but may be nothing).

remoteport - port to send the message to

id - ID to use for the message (a 32 bit number)

msgtype - NORMALMSG or EMERGENCYMSG

recptr - Pointer to a record to send

recsize - Type size IN BITS of the record

Result: The return of the send operation (see Accent Manual)

This routine packages up a simple Pascal record (i.e. no pointers, no ports) and sends it off to a process which will receive it using the RecRecord call.

RecRecord

Receives a Pascal record from a port.

Call:

```
function RecRecord(
                var localport           : Port;
                var remoteport          : Port;
                var id                  : long;
                var MsgType             : long;
                var recptr              : Pointer;
                var recsize             : integer)
        : GeneralReturn;
```

Parameters: *localport* - a port in the current process on which the message was received

remoteport - reply port if any

id - ID of the message (a 32 bit number)

msgtype - NORMALMSG or EMERGENCYMSG

recptr - Pointer to the received record

recsize - Type size IN BITS of the record

Result: Same as for Receive (see Accent Manual) plus BadMsgType

RecRecord receives a message containing a record, allocating space for the record automatically. It is meant to be used with SendRecord (above) and checks to see if the received message is compatible with the format used by SendRecord. If it is, the data pointer is placed in recptr and the other var parameters are updated. If the incoming message is bad in some way, either the error return from Receive will be returned or if the Receive succeeds but the message is of the wrong type, a BadMsgType will be returned. In all cases the last message received is kept in the exported LastRecMsg and can be examined there upon error.

14 OldTimeStamp: Convert between POS and Accent date formats

Implementer: David Golub

Abstract: OldTimeStamp is a compatibility module for converting between new time values and POS timestamps.

Files: oldtimestamp.pas, timedefs.pas

Exported Types

```
Internal_Time   = record     Accent time standard, GMT
   Weeks        : integer;   Number of weeks since 17-Nov-1858
   MSecInWeek   : long;      Number of milliseconds in that week
   end;

TimeStamp = packed record
   the fields in this record are ordered this way to optimize bits
   Hour:   0..23;
   Day:    1..31;
   Second: 0..59;
   Minute: 0..59;
   Month:  1..12;
   Year:   0..63;            year since 1980
   end;
```

OldCurrentTime

Gets current time as an old style time stamp.

Call: `function OldCurrentTime: TimeStamp;`

Result: TimeStamp for time, in system time zone.

NewToOldTime

Converts a new internal_time record to an old time stamp.

Call: `function NewToOldTime(NewTime: Internal_Time):TimeStamp;`

Result: Time stamp representing NewTime, in system time zone.

OldToNewTime

Converts an old time stamp to a new internal_time record.

Call: `function OldToNewTime(OldTime: TimeStamp): Internal_Time;`

Result: Internal time representing OldTime.

15 PascalInit:Process Initialization and Exporter of server ports

Implementers: Eugene Ball
Doug Philips
Michael Jones

Abstract: This module is used to complete the creation of a new process. It contains the first routine invoked in a process before the main program is called. It initializes the server environments and state variables for a process.

File: pascalinit.pas

Exported Types

Exported Variables:

```
UsrCmdLine          : CommandBlock; Command passed to user program

InPorts             : ptrPortArray; Ports inherited from parent
InPortsCnt          : long          Number of ports in InPorts

TimePort            : port;          Port to Time Server
SesPort             : port;          Port to Sesame Server
EMPort              : port;          Port to Environment Manager

PMPort              : port;          Port to Process Manager
NameServerPor       : port;          Port to Net message Server

UserTypescript      : Port;          This process' typescript
UserWindow          : Port;          This process' window
UserWindowShared    : Boolean;       TRUE if window shared with parent

TypescriptPort      : Port;          Only good for making new typescripts
SapphPort           : Port;          Only good for making new windows
```

Exported Exceptions:

```
Exception ExitProgram;
```
Anyone can raise this to exit a program and pass a SUCCESS return code to the program's caller. Handling this is dubious at best. Nothing in the boot file currently handles this exception.

```
Exception GRError( GR: GeneralReturn);
```
GRError is raised whenever a module wishes to signal a GeneralReturn error to be handled elsewhere. GR is the value of the GeneralReturn being signaled.

InitPascal

This is the first routine to run in an empty address space. It is called implicitly by Aload and should not ever be called by a user program.

Call: `Procedure InitPascal`

A spawn causes ALoad to initialize the new process state such that InitPascal will be the initial entry point of a pascal program when a resume is done. Spawn then does a Resume and InitPascal sets up the process state and calls the main program.

WARNING: It is critical that the routine number for InitPascal remain 0 for all time. Aload counts upon this fact.

InitProcess

This routine initializes a pascal process. This routine is exported mainly for use in spawn. It can only be used when there is an initial message to receive.

Call: `Procedure InitPascal (AmIClone : BOOLEAN)`

Parameters: *AmIClone* - Used to indicate whether the caller is a copy of some other process (fork, clone). True means caller was created with "Fork". False means caller was 'CreateProcess'ed.

How to modify the init message:

If you want to add a new kind of object to be passed to each process, do: For each thing you want to add, do the following: Add a XType field (of type 'TypeType') and an X field (of whatever type you are using). Add a CheckTypeType call in the corresponding place in the code below. After the CheckTypeType call add the code to extract the value just checked. That's all you have to do in this module. You have to then go into spawn and follow the directions there.

If you want to change the default set of ports passed to each process, do: For each port you want to add, do the following: Between 'FirstUserIndex' and the thing before it, add a constant name definition. When you are all done, make REAL sure that FirstUserIndex is 1 greater than the last named index. Add a statment to InitProcess that extracts the named port into wherever it goes. That's all you have to do in this module. You have to then go into spawn and follow the directions there.

DisablePrivs

Disable physical memory access and supervisor access for a process.

Call: `Function DisablePrivs( Proc: PORT): GeneralReturn;`

Parameters: *Proc* - The Kernel port of the process to affect.

Completion Code:

Success - Privileges have not been disabled.

Failure - Privileges not affected. This shouldn't ever occur unless you violate the precondition.

This procedure will only work if Proc is 'KERNELPORT' (i.e. you are modifying your own privileges), or if the process to be affected is suspended.

EnablePrivs

Enable physical memory access and supervisor access for a process.

Call: `Function EnablePrivs( Proc: PORT): GeneralReturn;`

Parameters: *Proc* - The Kernel port of the process to affect.

Completion Code:

Success - Privileges have been enabled.

Failure - Privileges not affected. This shouldn't ever occur unless you violate the precondition.

This procedure will only work if Proc is 'KERNELPORT' (i.e. you are modifying your own privileges), or if the process to be affected is suspended.

16 PathName: A Logical Name Interface to Sesame

Implementers: Michael B. Jones
David Golub

Abstract: The following routines provide a non-primitive interface to the Sesame File Server. These routines make use of both Environment Manager functions and Name Server functions. All *pathnames* should be handled by routines at this level.

Files: pathname.pas, sesamedefs.pas envmgrdefs.pas

Exported Types

APath_Name:

A full Name Server pathname string.

```
APath_Name          = string[Path_Name_Size];   An abs. pathname
Path_Name_Size      = 255;   Number of characters in a Path_Name

Entry_List          = ↑ Entry_List_Array;
Entry_List_Array    = array [0..0] of Entry_List_Record;   hack
Entry_List_Record:      ScanNames returns array of Entry_List_Record.
  Entry_List_Record     = record
    EntryName           : Entry_Name;
    EntryVersion        : long;
    EntryType           : Entry_Type;
    NameStatus          : Name_Status;
    end;
```

Entry_Type:

The kinds of objects which can be in the name data base.

```
Entry_Type          = 0 .. #77777;

Entry_All           = 0;      Special value referencing all entry types
Entry_File          = 1;      Entry_Data is a File_ID
Entry_Directory     = 2;      Name refers to another level of the
                                  name hierarchy.  Entry_Data is empty.
Entry_Port          = 3;      Entry_Data is a port
Entry_RESERVED      = 4 .. #377;  These values reserved for expansion
Entry_UserDefined   = #400 .. #77777;   Values available to the user
```

Env_Var_Name:

The name string for an environment variable. The syntax of the name is the same as for an arbitrary entry name in the name server.

```
Env_Var_Name            = string[Env_VarName_Size];
Env_VarName_Size    = Entry_Name_Size;
```

Extension_List:

a list of name extensions to tack onto a pathname (before any version number) when doing an extension search. Each extension in the list must be terminated by the semicolon (;) character.

```
Extension_List        = string[Extension_String_Size];
Extension_String_Size = 80;
```

File_Data: *A pointer to a file mapped into memory*

```
File_Data    = pointer;        A pointer to data for file calls
```

Name_Flags:

Flags giving desired treatment of names in Name Server calls.

Note that specific flag values may be illegal for certain calls, and must be zero.

```
Name_Flags       = 0 .. #3;
NFlag_Deleted    = #000001;   Allow deleted names
NFlag_NoNormal   = #000002;   Disallow normal (not deleted) names
NFlag_RESERVED   = #177774;   These bits reserved for expansion
```

Name_Status:

Flags useful for determining the disposition of a name in the name data base.

```
Name_Status      = 0 .. #7;

NStat_Deleted    = #000001;   Set if name is deleted
NStat_High       = #000002;   Set if name is highest undeleted version
NStat_Low        = #000004;   Set if name is lowest undeleted version p
NStat_RESERVED   = #177770;   These bits reserved for expansion
```

Path_Name: *An absolute, relative, or logical non-wild pathname.* **Wild_Path_Name:** *A potentially wild pathname.*

```
Path_Name           = string[Path_Name_Size];
Wild_Path_Name      = string[Path_Name_Size];
```

16.1 Interfaces to Sesame Calls

These calls accept either an absolute, relative or logical pathname. They should be used when a program does not necessarily have an absolute pathname.

ReadFile

Reading a file

Call:

```
function ReadFile(
            Var PathName                : Path_Name;
            Var Data                    : File_Data;
            Var ByteCount               : long)
        : GeneralReturn;
```

Parameters: *PathName* - Path name of the file to be read. Is returned set to the absolute pathname of the file read.

Data - Is set to a pointer to the returned data.

ByteCount - Is set to number of bytes read.

Completion Code:

success - the data was successfully mapped into memory

NameNotFound - no entry was found with the name *PathName*

NotAFile - the entry found with the name *PathName* was not a file

EnvVariableNotFound - The default search list was incorrect.

The *ReadFile* call is equivalent to calling *FindFileName* and then using the returned absolute pathname in a call to *SubReadFile*.

ReadExtendedFile

Reading one of a list of files

Call:

```
function ReadExtendedFile(
            Var PathName                : Path_Name;
                ExtensionList           : Extension_List;
                ImplicitSearchList      : Env_Var_Name;
            Var Data                    : File_Data;
            Var ByteCount               : long)
        : GeneralReturn;
```

Parameters:

PathName - The pathname of the file to be read. Returned set to the absolute pathname of the file read.

ExtensionList - A list of possible filename extensions.

ImplicitSearchList - Name of the search list to use if a partial pathname is supplied. If blank, the list "Default" is used.

Data - Set to point to the returned data

ByteCount - Set to number of bytes read

Completion Code:

success - the data was successfully mapped into memory

NameNotFound - no entry was found under apathname

NotAFile - the entry found under apathname was not a file

EnvVariableNotFound - The Implicit or Default search list was incorrect.

The *ReadExtendedFile* call is equivalent to calling *FindExtendedFileName* followed by a call to *SubReadFile* with the returned absolute pathname with extension.

WriteFile

Writing a file

Call:

```
function WriteFile(
            Var PathName                : Path_Name;
                Data                    : File_Data;
                ByteCount               : long)
        : GeneralReturn;
```

Parameters: *PathName* - Path name of the file to be written. Is returned set to the absolute pathname of the file written.

Data - Pointer to the data to be written.

ByteCount - Number of bytes to write.

Completion Code:

success - the data was written under the name returned

InvalidVersion - the explicit version number was less than or equal to that of an existing version

The *WriteFile* call is equivalent to calling *ExpandPathName* followed by a call to *SubWriteFile* with the returned absolute pathname.

16.2 Name searching routines

CompletePathName

Do file-name completion on the filename indicated by WildPathName.

Call:

```
function CompletePathName(
            var WildPathName            : Wild_Path_Name;
                ImplicitSearchList      : Env_Var_Name;
                FirstOnly               : boolean;
            var Cursor                  : integer)
        : long;
```

Parameters: *WildPathName* - The partial filename to be expanded. Changed to indicate the (partially) completed filename.

ImplicitSearchList - Name of the search list to use if a partial path name is supplied.

FirstOnly - If true, only look in the first item of the search list. Should ordinarily be false.

Cursor - Position in WildPathName AFTER which the implicit '*' should be inserted. Changed to indicate the corresponding position in the expanded

WildPathName. For now, the corresponding position is FORCED to be at the END of an entry name. Most common usage is Cursor = length(WildPathName).

Result: number of names that matched WildPathName. 0 => no matches at all; WildPathName unchanged 1 => unique match; WildPathName is the expanded name. n => several matches; WildPathName is the part that matches them all.

CompletePathName finds the number of names that will match a pathname with some wildcard component. *WildPathName* is taken relative to *ImplicitSearchList* or if that is blank relative to "Default".

ExpandPathName

Expanding a pathname into an absolute pathname

Call:

```
function ExpandPathName(
            Var PathName                : Wild_Path_Name;
                ImplicitSearchList      : Env_Var_Name)
        : GeneralReturn;
```

Parameters: *PathName* - the (potentially) relative pathname to be expanded. Returned set to absolute pathname

ImplicitSearchList - the *pathname* is to be interpreted as relative to this logical name. If it is blank, the logical name "Default" is used.

Completion Code:

success - the name search was successful

EnvVariableNotFound - the specified logical name was not defined

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

BadName - *PathName* was incorrectly formatted.

The *ExpandName* call accepts a pathname relative to an *implicit logical name*, and returns the corresponding absolute pathname after the logical name is expanded.

If no implicit logical name is specified in the call and the *pathname* is a relative pathname, then the expansion is done relative to the logical name "Default" (resulting in an expansion in the *current directory*). The user may of course, override the default expansion by specifying either an absolute pathname or an explicit logical name in the *pathname* parameter. Logical names will be recursively flattened as necessary to complete the expansion. Undefined logical names will cause the expansion to fail with an *EnvVariableVarNotFound* error.

This call differs from *FindName* in that only a macro expansion is done, and no attempt is made to verify that the name previously existed or is even valid. Whereas *FindName* should normally be used before a *LookupName* type of operation, *ExpandName* should normally be used before an *EnterName* type of operation. Note that only the first element in the logical name search list is used by this call.

FindPathName

Performing a name search for any type of name

Call:

```
function FindPathName(
            Var PathName                : Path_Name;
                ImplicitSearchList      : Env_Var_Name;
                FirstOnly               : boolean;
            Var EntryType               : Entry_Type;
            Var NameStatus              : Name_Status)
          : GeneralReturn;
```

Parameters:

PathName - the relative, absolute, or logical pathname to be searched for. Is returned set to the absolute pathname that was found.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If it is blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

EntryType - Set to the type value of the entry found

NameStatus - Set to low version or high version

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

The *FindPathName* call searches for a pathname relative to the *implicit logical name*, returning the name found and the entry type associated with it. If no logical name is specified in the call and the *pathname* is a relative pathname, then the search is done using the logical name "Default". The user may of course, override the *implicit logical name* given in the call by specifying either an absolute pathname or an explicit logical name in the *pathname* parameter. Logical names will be recursively expanded as necessary to complete the search. Except for the case that the top-level logical name is undefined, errors in the search will not terminate it. Thus, if a name in a list is undefined, it will be effectively ignored.

In summary, there are three cases of *pathnames*:

1. **absolute pathname** -- requires only a *TestName* on the name to get the entry type.

2. **relative pathname** -- requires the flattening of the *implicit logical name*, and then iterating down the search list testing for *pathname* with *TestName* until it is found.

3. **logical pathname** -- requires the parsing off the logical name part, the flattening of the logical name, and then iterating down the search list testing for *pathname* with *TestName* until it is found.

FindFileName

Performing a name search for a file name

Call:

```
function FindFileName(
            Var PathName                : Path_Name;
                ImplicitSearchList      : Env_Var_Name;
                FirstOnly               : boolean)
        : GeneralReturn;
```

Parameters:

PathName - the relative, absolute, or logical pathname to be searched for. Is returned set to the absolute pathname that was found.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If it is blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

NotAFile - the name did not refer to a file

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

Is the same as *FindPathName* but only looks for entries of type file

FindExtendedPathName

Performing a name search with extensions

Call:

```
function FindExtendedPathName(
            Var PathName                : Path_Name;
                ExtensionList           : Extension_List;
                ImplicitSearchList      : Env_Var_Name;
```

```
        FirstOnly                : boolean;
    Var EntryType                : Entry_Type;
    Var NameStatus               : Name_Status)
  : GeneralReturn;
```

Parameters: *PathName* - the relative, absolute, or logical pathname to be searched for. Changed to the name actually found.

ExtensionList - The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

NameStatus - Set to low version or high version

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

The *FindExtendedPathName* call is like the *FindPathName* call with the additional function that it performs a two-dimensional search: first down the extension list and then down the directory search list. Each extension string is successively concatenated to the terminal component of the name and tried, successively in each search list directory (if an absolute pathname was not specified) until a match is found.

FindExtendedFileName

Performing a file name search with extensions

Call:

```
function FindExtendedFileName(
            Var PathName             : Path_Name;
                ExtensionList        : Extension_List;
                ImplicitSearchList   : Env_Var_Name;
                FirstOnly            : boolean)
        : GeneralReturn;
```

Parameters: *PathName* - the relative, absolute, or logical pathname to be searched for. Changed to the name actually found.

ExtensionList - The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension. An empty list means that no extensions are applied.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

NotAFile - name was found but was not a file

The *FindExtendedFileName* is the same as the *FindExtendedPathName* except that it only searches for names of type file.

FindTypedName

Performing a name search for any specific type of name with an optional list of extensions

Call:

```
function FindTypedName(
            Var PathName              : Path_Name;
                ExtensionList         : Extension_List;
                ImplicitSearchList    : Env_Var_Name;
                FirstOnly             : boolean;
            Var EntryType             : Entry_Type;
            Var NameStatus            : Name_Status)
        : GeneralReturn;
```

Parameters:

PathName - the relative, absolute, or logical pathname to be searched for. Is returned set to the absolute pathname that was found.

ExtensionList - The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If it is blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

EntryType - Entry type being searched for. *Entry_all* finds the first one, and then sets EntryType to the type found.

NameStatus - Set to low version or high version

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

ImproperEntryType - name was found but was of wrong type

The *FindTypedName* call is like the *FindExtendedPathName* call except that it searches for a specific type of name. The extension list may be empty.

FindWildPathnames

Finding all the matches for a wild-carded name using a search list

Call:

```
function FindWildPathnames(
            Var WildPathName          : Path_Name;
                ImplicitSearchList    : Env_Var_Name;
                FirstOnly             : boolean;
                NameFlags             : Name_Flags;
                EntryType             : Entry_Type;
            Var FoundInFirst          : boolean;
            Var DirName               : APath_Name;
            Var EntryList             : Entry_List;
            Var EntryListCnt          : long)
        : GeneralReturn;
```

Parameters:

WildPathName - the relative, absolute, or logical pathname to be searched for. Only the terminal name component may have wild-card characters. Is returned set to the absolute pathname that was found.

ImplicitSearchList - if *pathname* was a relative pathname then the *pathname* is to be interpreted as relative to this search list. If it is blank, the logical name "Default" is used.

FirstOnly - if set to *first*, only the first name in the search list will be used; if set to *full*, a complete search will be performed.

EntryType - the type value of the entries to return. *Entry_all* causes all types of entries to be returned

FoundInFirst - Returned TRUE if and only if the first item in the search list produced the match.

DirName - Returned set to the absolute pathname of the directory in which the matches were found

EntryList - List of entry name, types, version and status as returned by *SesScanNames*

EntryListCnt - Count of entries in *EntryList*

Completion Code:

success - the name search was successful

NameNotFound - the specified name was not found

EnvVariableNotFound - the specified logical name was not defined

BadName - *PathName* was incorrectly formatted.

SearchLoopList - a recursively defined logical name resulted in a logical name expansion loop

Finds matches for a wildcarded name using a search list. For each item in the search list, the name is looked up using *SesScanNames*. If any matches are found, the search stops and the results from SesScanNames are returned. Note that this routine may not be quite what you want -- it returns first non-empty match-set, NOT the union of all the match-sets. Think about the distinction before deciding whether this routine is appropriate for your application!

16.3 Name manipulation routines

ExtractSimpleName

Finding the terminal component and version number of a pathname.

Call:

```
procedure ExtractSimpleName(
                Name                    : Path_Name;
            Var StartTerminal           : integer;
            Var StartVersion            : integer);
```

Parameters:

Name - The pathname to check

StartTerminal - Returns the index of the first character of the terminal component of the name. Will be length(Name) + 1 if the name is a directory name (ends with '>').

StartVersion - Returns the index of the version suffix of name (at the ';'). Will be length(Name) + 1 if the name has no version.

The *ExtractSimpleName* call returns the indices of the terminal component of the name and the version number.

SimpleName

Return the terminal versionless component of a pathname.

Call:

```
function SimpleName(
                    PathName                : Path_Name)
                : Entry_Name;
```

Parameters: *PathName* - The pathname from which to extract the terminal node.

Result: The terminal component of PathName.

Returns the terminal node of a name without a version number.

StripCurrent

Contracting an absolute pathname to a relative pathname when possible

Call:

```
function StripCurrent(
                Var WildPathName            : Wild_Path_Name)
                : GeneralReturn;
```

Parameters: *WildPathName* - the absolute pathname to be converted

Completion Code:

success - the absolute pathname was processed correctly

EnvVariableNotFound - the logical name "Current" was undefined

Failure - The initial components of *WildPathName* are not the same as "Current".

BadName - *PathName* was incorrectly formatted.

This routine may be called by an application in order to try to shorten an absolute pathname returned by the Name Server into a relative pathname suitable for typeout to the user. If the initial path components of *WildPathName* matched the first entry of the logical name "Current", then the relative pathname with the initial components removed is returned. Otherwise, the absolute pathname is returned. A version number is included.

AddExtension

Add an extension name to a pathname

Call:

```
Procedure AddExtension(
                Var FileName                : Path_Name;
                    Extension               : String);
```

Parameters: *FileName* - Name to check. It is changed if the extension is added.

Extension - The extension to add

Adds an extension to a file name if it is not already there.

ChangeExtension

Change an extension of a pathname

Call:
```
Procedure ChangeExtensions(
            Var Name                    : Path_Name;
                EList                   : Extension_List;
                NewExt                  : string);
```

Parameters: *Name* - The path name to be modified.

EList - The list of extensions to check for and replace.

NewExt - The extension to add.

The *ChangeExtension* call removes any of a list of extensions from a file name if they are there and then adds NewExt to the end.

NextExtension

Finding the next extension in a list.

Call:
```
function NextExtension(
            Var EList               : Extension_List)
          : string;
```

Parameters: *Elist* - the list of extensions to be modified

Result: the next extension

The *NextExtension* call retrieves the next extension from a list of them which are terminated by semicolons. Removes the extension from the list.

RemoveExtension

Removing extensions

Call:
```
Procedure RemoveExtension(
            Var FileName            : Path_Name;
                Extension           : String);
```

Parameters: *FileName* - The file name to check. It is altered to remove the extension if it is there.

Extension - The extension to remove.

The *RemoveExtension* removes an extension from a file name if it is there.

Index1Unquoted

Find an unquoted character in a pathname.

Call:

```
function Index1Unquoted(
                S                       : Wild_Path_Name;
                C                       : char)
        : integer;
```

Parameters: *S* - The string to search

C - The character to find; it should not be ""

Result: The index of the matching character, or zero if none.

Finds the first occurrence of *C* in *S* that is not preceded by a ' character.

IsQuotedChar

Look for quoted characters in a pathname

Call:

```
function IsQuotedChar(
                S                       : Wild_Path_Name;
                Index                   : integer)
        : boolean;
```

Parameters: *S* - The string to check.

Index - The index of the character to be checked.

Completion Code:

true - the character is quoted by preceding ' characters.

false - the character is not quoted.

Checks whether *S[Index]* is preceded by a ' character.

17 PMatch: Pattern Matching Routines

Implementers: Gene Ball
Michael B. Jones
Brad Meyers

Abstract: Does pattern matching on strings. Patterns accepted are as follows:

"*" matches 0 or more characters.

"?" matches exactly 1 character.

"*" matches '*', other pattern chars can be quoted also.

File: pmatch.pas

Exported Types

```
pms255 = String[255];
```

Exported exceptions

`Exception BadPatterns;` Raised if outPatt and inPatt do not have the same patterns in the same order for PattMap

PattDebug

Sets the global debug flag

Call:
```
procedure PattDebug(v              : boolean)
```

Parameters: *v* - value to set debug to.

Changes debug value.

IsPattern

Tests to see whether str contains any pattern matching characters.

Call:
```
function IsPattern(
                    str            : pms255)
          : boolean
```

Parameters: *str* - string to test.

Result: true if *str* contains any pattern matching characters, otherwise false.

PattMatch

Compares str against pattern.

Call:

```
function PattMatch(
            var str, pattern            : pms255)
        :boolean
```

Parameters: *str* - full string to compare against pattern.

pattern - pattern to compare against. It can have special characters in it.

Result: true if string matches pattern, otherwise false.

PattMap

Compares** str **against** inpatt**, putting the parts of Str that match** Inpatt **into the corresponding places in Outpatt and returning the result.

Call:

```
function PattMap(
            var str,inpatt,outpatt,outstr      :pms255;
                fold                          :boolean)
        :boolean
```

Parameters: *str* - is full string to compare against pattern.

inpatt - is pattern to compare against. It can have special characters in it.

outpatt - pattern to put the parts of str into; it must have the same special characters in the same order as in inpatt

outStr - the resulting string if PattMap returns true

fold - if true, upper and lower cases match, otherwise they are distinct.

Result: True if the string matches the pattern; false otherwise

Exceptions: *BadPatterns* - if *outpatt* and *inPatt* do not have the same patterns in the same order.

EXAMPLES:

```
PattMap('test9.pas', 'test'0.pas', 'xtest'0.pas', outstr, TRUE)
   returns TRUE, with outstr = 'xtest9.pas'

PattMap('test9.pas', '*.pas', '*.ada', outstr, FALSE)
   returns TRUE, with outstr = 'test9.ada'
```

18 RealFunctions - Standard functions for reals.

Implementer: John Strait

Abstract: RealFunctions implements many of the standard functions whose domain and/or range is the set of real numbers. The implementation of these functions was guided by the book **Software Manual for the Elementary Functions**, William J. Cody, Jr. and William Waite, (C) 1980 by Prentice-Hall, Inc.

The domain (inputs) and range (outputs) of the functions are given in their abstract. The following notation is used. Parentheses () are used for open intervals (those that do not include the endpoints), and brackets [] are used for closed intervals (those that do include their endpoints). The closed interval [RealMLargest, RealPLargest] is used to mean all real numbers, and the closed interval [-32768, 32767] is used to mean all integer numbers.

Currently all functions described by Cody and Waite are implemented.

DISCLAIMER:

Only the most cursory testing of these functions has been done. No guarantees are made as to the accuracy or correctness of the functions. Validation of the functions must be done, but at some later date.

Design: AdX, IntXp, SetXp, and Reduce are implemented as Pascal functions. It is clear that replacing the calls with in-line code (perhaps through a macro expansion) would improve the efficiency.

Many temporary variables are used. Elimination of unnecessary temporaries would also improve the efficiency.

Many limit constants have been chosen conservatively, thus trading a small loss in range for a guarantee of correctness. The choice of these limits should be re-evaluated by someone with a better understanding of the issues.

Some constants are expressed in decimal (thus losing the guarantee of precision). Others are expressed as Sign, Exponent, and Significand and are formed at execution time. Converting these two 32-bit constants which are Recast into real numbers would improve the correctness and efficiency.

More thought needs to be given to the values which are returned after resuming from an exception. The values that are returned now are the ones recommended by Cody and Waite. It seems that Indefinite values (NaNs in the IEEE terminology) might make more sense in some cases.

Sqrt

Compute the square-root of a number.

Call: `function Sqrt( X : Real ): Real`

Result: Square-root of *X*.

Domain = [0.0, RealPLargest]. Range = [0.0, Sqrt(RealPLargest)].

Ln

Compute the natural log of a number.

Call: `function Ln( X : Real ): Real`

Result: Natural log of *X*.

Domain = [0.0, RealPLargest]. Range = [RealMLargest, Ln(RealPLargest)].

Log10

Compute the log to the base 10 of a number.

Call: `function Log10( X : Real ): Real`

Result: Log to the base 10 of *X*.

Domain = [0.0, RealPLargest]. Range = [RealMLargest, Log10(RealPLargest)].

Exp

Compute the exponential function.

Call: `function Exp( X : Real ): Real`

Result: *e* raised to the *X* power.

Domain = [-85.0, 87.0]. Range = (0.0, RealPLargest].

Power

Call: `function Power( X, Y : Real ): Real`

Result: *X* raised to the *Y* power.

Compute the result of an arbitrary number raised to an arbitrary power. DomainX = [0.0, RealPLargest]. DomainY = [RealMLargest, RealPLargest]. Range = [0.0, RealPLargest]. Restrictions: 1) if X is zero, Y must be greater than zero. 2) X raised to the Y is a representable real number.

PowerI

Call:
```
function PowerI(
                    X       : Real;
                    Y       : Integer )
          : Real;
```

Result: X raised to the Y power.

Compute the result of an arbitrary number raised to an arbitrary integer power. The difference between Power and PowerI is that negative values of X may be passed to PowerI. DomainX = [RealMLargest, RealPLargest]. DomainY = [-32768, 32767]. Range = [RealMLargest, RealPLargest]. With the restrictions that 1) if X is zero, Y must be non-zero. 2) X raised to the Y is a representable real number.

Sin

Compute the sin of a number.

Call:
```
function Sin( X : Real ): Real
```

Result: Sin of X.

Domain = [-1E5, 1E5]. Range = [-1.0, 1.0].

Cos

Compute the cosin of a number.

Call:
```
function Cos( X : Real ): Real
```

Result: Cos of X.

Domain = [-1E5, 1E5]. Range = [-1.0, 1.0].

Tan

Compute the tangent of a number.

Call:
```
function Tan( X : Real ): Real
```

Result: Tangent of X.

Domain = [-6433.0, 6433.0]. Range = [RealMInfinity, RealPInfinity].

CoTan

Compute the cotangent of a number.

Call:
```
function CoTan( X : Real ): Real
```

Result: Cotangent of X.

Domain = [-6433.0, 6433.0]. Range = [RealMInfinity, RealPInfinity].

ArcSin

Compute the arcsin of a number.

Call: `function ArcSin( X : Real ): Real`

Result: Arcsin of *X*.

Domain = [-1.0, 1.0). Range = [-Pi/2, Pi/2). It seems that the Domain and Range ought to be closed intervals, however this implementation apparently returns a number very close to zero when X is 1.0, rather than returning Pi/2 as it should.

ArcCos

Compute the arccosin of a number.

Call: `function ArcCos( X : Real ): Real`

Result: Arccosin of *X*.

Domain = (-1.0, 1.0]. Range = (-Pi/2, Pi/2]. It seems that the Domain and Range ought to be closed intervals, however this implementation apparently returns a number very close to zero when X is -1.0, rather than returning -Pi/2 as it should.

ArcTan

Compute the arctangent of a number.

Call: `function ArcTan( X : Real ): Real`

Result: Arctangent of *X*.

Domain = [RealMLargest, RealPLargest]. Range = (-Pi/2, Pi/2). Seems fine except for very large numbers.

ArcTan2

Compute the arctangent of the quotient of two numbers.

Call: `function ArcTan2( Y, X : Real ): Real`

Result: Arctangent of *Y / X*.

Seems fine except for very large Y/X. One interpretation is that the parameters represent the cartesian coordinate (X,Y) and ArcTan2(Y,X) is the angle formed by (X,Y), (0,0), and (1,0). DomainY = [RealMLargest, RealPLargest]. DomainX = [RealMLargest, RealPLargest]. Range = [-Pi, Pi].

SinH

Compute the Hyperbolic Sine of a number.

Call: `SinH( x: real ) : real`

Result: The Hyperbolic Sine of X

Domain = [-87.33, 87.33]. Range = [RealMLargest, RealPLargest].

CosH

Compute the Hyperbolic Cosine of a number.

Call: `function CosH (x: real) :real`

Result: The Hyperbolic Cosine of X

Domain = [-87.33, 87.33] Range = [1.0, RealPLargest].

TanH

Compute the Hyperbolic Tangent of a number.

Call: `function TanH( x: real) : real`

Result: The Hyperbolic Tangent of X.

Domain = [-8.66433975625,8.66433975625]. Range = [-1.0, 1.0].

19 SaltError: Translation of error codes

Implementers: Eugene Ball
Michael B. Jones
Amy Butler

Abstract: SaltError is the standard system error module. It provides a number of facilities for generating and printing error messages.

There are three classes of errors :

a) Warnings - informative, preceeded by a single "*".

b) Errors - indicating a true error, preceeded by "**". It may be possible to recover from an Error. The recovery is left to the application program.

c) Fatal Errors - No recovery possible, also preceeded by "**".

File: salterror.pas

Exported Types

```
GeneralReturn = integer;     Values returned from system calls

GR_Error_Type = (GR_Warning, GR_Error, GR_FatalError);
```

GRWriteStdError

Takes the GR value, finds the message and writes it.

Call:
```
Procedure GRWriteStdError(
                GR              : GeneralReturn;
                ER_Type         : GR_Error_Type;
                InMsg           : PString)
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

InMsg - An optional message to display.

GRStdError

Takes The GR value, finds the message and returns it in OutMsg.

Call:
```
Procedure GRStdError(
                GR      : GeneralReturn;
                ER_Type     : GR_Error_Type;
                InMsg       : PString;
            var OutMs       : PString)
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

InMsg - An optional message to display.

OutMsg - The translated error msg

If fatal error, outmsg will be lost.

GRWriteErrorMsg

Takes the GR value, finds the message and writes it as *<**stars*** Progname : Module: GrMessage.>

Call:

```
Procedure GRWriteErrorMsg(
                GR                : GeneralReturn;
                ER_Type           : GR_Error_Type;
                ProgName          : String;
                InMsg             : PString)
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

ProgName - Program name to display - provides information about where in the system, program and module the error occured.

InMsg - An optional message to display.

This routine is used for errors that may not be a direct result of user action.

GRErrorMsg

Takes the GR value, finds the message and returns it in OutMsg as *<**stars*** Progname: Module: GrMessage.>

Call:

```
Procedure GRErrorMsg(
                  GR              : GeneralReturn;
                  ER_Type         : GR_Error_Type;
                  ProgName        : String;
                  InMsg           : PString;
              var OutMsg          : PString)
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

ProgName - Program name to display - provides information about where in the system, program and module the error occured.

InMsg - An optional message to display.

OutMsg - The translated error msg

This routine is used for errors that may not be a direct result of user action.

ErrorMsgPMBroadcast

Takes the GR value, finds the message, and prints the message in the process manager window.

Call:

```
Procedure ErrorMsgPMBroadcast(
                    GR              : GeneralReturn;
                    ER_Type         : GR_Error_Type;
                    ProgName        : String;
                    InMsg           : PString)
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

ProgName - Program name to display - provides information about where in the system, program and module the error occured.

InMsg - An optional message to display.

GRStdErr

Takes the GR value, finds the message and returns it in OutMsg.

Call:

```
Function GRStdErr(
                    GR              : GeneralReturn;
                    ER_Type         : GR_Error_Type;
                    InMsg           : PString;
                var OutMsg          : PString)
            : boolean;
```

Parameters: *GR* - The return code to translate.

ER_Type - Warning, Error or Fatal Error

InMsg - An optional message to display.

OutMsg - The translated error msg

If FatalError, outmsg will be lost.

20 Spawn: Create and initialize a new process

Implementers: Eugene Ball
Mary R. Thompson
Doug Philips
William Maddox
David Golub
Michael B. Jones

Abstract: Create a new process and notify the Process Manager of its existence. An initialization message containing state and system server ports is sent to be received in the new process (by **InitProcess** in **PascalInit**). Spawn is used to handle the general case of process creation while Exec and Split are implemented by calling Spawn with specific arguments.

Files: spawn.pas, spawninitflags.pas

Exported Types

```
CmdLineString          = STRING[ 255];

ConnectionInheritance = (NewOne, Given, GivenReg);
```

NewOne - Make a new connection.
For Sapphire, this means create a newwindow and typescript.
For file system, this means duplicate ours.
Given - Use the ports given in the call.
The parent process retains ownership.
GivenReg - Use the ports given in the call.
The Child process will own the ports.

Exec

Create a new process running a new program. (Note: Unix Exec runs a new program in the existing process.)

Call:

```
function Exec(
                VAR ChildKPort          : Port;
                VAR ChildDPort          : Port;
                    ProcessName         : STRING;
                    HisCommand          : CommandBlock)
        : GeneralReturn;
```

Parameters: *ChildKPort* - set to the new process's kernel port.

ChildDPort - set to the new process's data port.

ProcessName - The name of the file to execute, and the name that will be registered with the Process Manager.

HisCommand - Used to set the new process's UsrCmdLine.

Completion Code:

Success - Process was created and loaded.

Failure - No new process was created.

This is the simplist way to run a program in a new process. The new process will share the caller's window and typescript and copy the caller's Environment Manager connection. The process will be registered with the Process Manager under its run file name (i.e. *ProcessName*). It will inherit its protection from the caller. It won't startup in the debugger.

Split

Create a copy of the current process. (Unix: fork)

Call:

```
function Split(
                var ChildKPort          : Port;
                var ChildDPort          : Port)
        : GeneralReturn;
```

Parameters: *ChildKPort* - set to the new process's kernel port.

ChildDPort - set to the new process's data port.

Completion Code:

IsParent - The original process.

IsChild - The copy.

This is the simplest way to 'fork' and have all the standard initialization done. The caller's window and typescript will be shared. The caller's Environment Manager state will be duplicated. The child will have the same protection as the parent.

Spawn

General case of process creation. This can exec as well as fork.

The Spice kernel calls to create processes, Fork and CreateProcess, do not pass state or ports to the new process. Spawn gets around this restriction by passing a message to the new process containing this information. InitProcess receives this message in the new process.

Basically Spawn just passes the system ports except if SaphConn of EMConn is "NewOne". In this case, it must create a new Sapphire and Environment connection. The Process Manager is notified (PMRegisterProcess) of the new process, its window, environment, and these are the rights to access physical memory and I/O device registers. The new process will be resumed or left for debugging (PMMakeDebugProcess) depending on the debugit parameter.

Call:

```
function Spawn(
                var ChildKPort    : Port;
                var ChildDPort    : Port;
```

```
                ProgName      : APathName;
                ProcName      : string;
                HisCommand    : CommandBlock;
                DebugIt       : boolean;
                ProtectChild  : boolean;
                SapphConn     : ConnectionInheritance;
                pWindow       : Port;
                pTypeScript   : Port;
                EMConn        : ConnectionInheritance;
                pEMPort       : Port;
                PassedPorts   : ptrPortArray;
                NPorts        : long;
                LoaderDebug   : BOOLEAN)
        : GeneralReturn;
```

Parameters:

ChildKPort - will be set to the Child's KernelPort in the Parent.

ChildDPort - will be set to the Child's DataPort in the Parent.

ProgName - the name of the .RUN file that you want loaded and executed. If this is null, a fork is done and the new process continues to run the existing program.

ProcName - the name of the child process as it will should appear in a SYSTAT, usually the same as ProgName.

HisCmdLine - this will be used to set the child's UserCommand.

DebugIt - if true, the new process is suspended and Mace is invoked on it befor it gets control.

ProtectChild - if true, the child process will not have access to physical memory or the ability to do I/O. If false, the child will inherit the parent's access capabilities.

SapphConn - If 'Given', the child's window and typescript will be taken from the pWindow and pTypeScript arguments. They will be shared (i.e. WindowShared = true) with these ports. 'GivenReg' has the semantics above and ownership rights for these two ports are passed to the child. If 'NewOne', Sapphire will be asked to create a new window and TypeScript will be initialized to this window. These new ports are passed with ownership rights.

pWindow - a window to share with the new process.

pTypeScript - a TypeScript to share with the new process.

EMConn - this controls access to the environment manager. If 'Given' then the connection will be taken from 'pEMPort'. 'GivenReg' is the same as 'Given', except that ownership rights to the port are passed. If 'Newone' then a new Environment Manager connection will be made, copying the state of the current connection.

pEMPort - parent's Environment Manager port.

PassedPorts - an array of ports to pass to the child. This contains only the ports above and beyond any system ports that you wnat to pass to the child, and will be available in Inports, with the same indexes as in this array. Everything else that you used to have to set by hand is now controlled by other parameters to spawn.

NPorts - the number of ports being passed in PassedPorts.

LoaderDebug - turns on Spawn and loader debugging.

Completion Code:

IsParent - In the fork case

IsChild - In the fork case

Success - In the exec case

Failure - In the exec case

If the ProgName string is not empty, Spawn performs the Exec function by creating a new process and Aloading into it.

If you don't specify 'newone' as the SapphConn parameter, you will have to do a EnableWinListener on the window yourself (if the parent hasn't already done it). When 'newone' is used, Spawn will make a window and typescript and set up the typescript's keytranslation table for you.

21 Spice_String: PERQ String hacking routines.

Implementers:

Don Scelza
Joseph M. Newcomer
J.G. Chandler

Abstract: This module implements SAIL-like string hacking routines for the Three River PERQ Pascal. It is a complete replacement for PERQ_String and Sail_String.

Strings in PERQ Pascal are stored a single character per byte with the byte indexed by 0 being the length of the string.

File: spice_string.pas

Exported Types

```
BreakKind = set of BreakType;
BreakType = (Append,Retain,Skip,
         FoldUp,FoldDown,
         Inclusive,Exclusive);

BreakTable = ↑ BreakRecord;
BreakRecord = record
  Breakers: set of Char;
  Omitters: set of Char;
  Flags: BreakKind
end;

PString = String[MaxPStringSize];
MaxPStringSize=255;          Length of strings
Inf = -32742;                magic value decoded as length-of-string
```

Exceptions Exported

```
Exception StrBadParm(FuncName, StringArgument: PString; ParmValue: integer;)
```

Abstract — Raised when inconsistent (bad index or length) parameters are passed to procedures.

Parameters

FuncName	Name of function. (Is all upper case.)
StringArgument	The string paramenter with which the illegal operation was to be performed.
Parmvalue	An illegal value. (In some instances, the combination of two parameters is illegal; for these, one of the parameters is arbitarily chosen.)

Resume — If the exception returns, the procedures will exit immediately, returning meaningless results.

Raised by Adjust, Pad, SetBreak

```
Exception StrLong; declared in module Except
```

Abstract: A result string will be too long.

Raised by ConCat, Cat3, Cat4, Cat5, Cat6, InsertChars, SubstrTo, SubstrFor

Resume If the exception returns, the result will be truncated to 255 characters.

Adjust

This procedure is used to change the dynamic length of a string.

Call:
```
procedure Adjust(
                    var Str         :PString;
                        Len         :Integer)
```

Parameters: *Str* - is the string that is to have the length changed.

Len - is the new length of the string. This parameter must be no greater than MaxPStringSize.

Result: This procedure does not return a value.

Exceptions: *StrBadParm* - If Len > MaxPStringSize or less than 0. If the error resumes, Str is not modified.

AppendChar

puts c on the end of Str

Call:
```
procedure AppendChar(
                    var Str         : PString;
                        c           : Char)
```

Parameters: *Str* - is the left String and c goes on the end.

Exceptions: *StrBadParm* - As this calls Adjust, StrBadParm will be raised if length (Str) is equal to MaxPStringSize.

AppendString

puts Str2 on the end of Str1

Call:
```
procedure AppendString(
                    var Str1                : PString;
                        Str2                : PString)
```

Parameters: *Str1* - is the left String and Str2 goes on the end.

Modifies Str1.

Cat3

Concatenates three strings together.

Call:
```
function  Cat3(
                  Str1,Str2,Str3          : PString)
          : PString
```

Parameters: *Str1,Str2,Str3* - Strings to concatenate.

Result: Returns a single string composed of the parameters.

Exceptions: *StrLong* - If the total length is greater then MaxPStringSize then raise StrLong exception. If resume from the exception, the result is truncated to 255 characters. This exception is tested as each string is appended, so it may occur multiple times for one call on Cat-n.

Uses ConCat to combine the arguments into a temporary.

Cat4

Concatenates four strings together.

Call:
```
function  Cat4(
                  Str1,Str2,Str3,Str4             : PString)
          : PString
```

Parameters: *Str1,Str2,Str3,Str4* - Strings to concatenate.

Result: Returns a single string composed of the parameters.

Exceptions: *StrLong* - If the total length is greater then MaxPStringSize then raise StrLong exception. If resume from the exception, the result is truncated to 255 characters. This exception is tested as each string is appended, so it may occur multiple times for one call on Cat-n.

Uses ConCat to combine the arguments into a temporary.

Cat5

Concatenates five strings together.

Call:
```
function  Cat5(
                  Str1,Str2,Str3,Str4,Str5        : PString)
          : PString
```

Parameters: *Str1,Str2,Str3,Str4,Str5* - Strings to concatenate.

Result: Returns a single string composed of the parameters.

Exceptions: *StrLong* - If the total length is greater then MaxPStringSize then raise StrLong exception. If resume from the exception, the result is truncated to 255 characters. This exception is tested as each string is appended, so it may occur multiple times for one call on Cat-n.

Uses ConCat to combine the arguments into a temporary.

Cat6

Concatenates six strings together.

Call:
```
function  Cat6(
                  Str1,Str2,Str3,Str4,Str5,Str6    : PString)
          : PString
```

Parameters: *Str1,Str2,Str3,Str4,Str5,Str6* - Strings to concatenate.

Result: Returns a single string composed of the parameters.

Exceptions: *StrLong* - If the total length is greater then MaxPStringSize then raise StrLong exception. If resume from the exception, the result is truncated to 255 characters. This exception is tested as each string is appended, so it may occur multiple times for one call on Cat-n.

Uses ConCat to combine the arguments into a temporary.

Concat

Concatenates two strings together.

Call:
```
function  Concat(
                  Str1,Str2        : PString)
          : PString
```

Parameters: *Str1,Str2* - the two strings that are to be concatenated.

Result: Returns a single string as described by the parameters.

Exceptions: *StrLong* - If Length(Str1) + Length(Str2) is greater then MaxPStringSize then raise StrLong exception. If control is resumed from the exception, concat will return a string truncated to length MaxPStringSize.

Uses MVBW (Move bytes Q-code) to copy Str2 onto the end of Str1.

ConvUpper

Converts str to all upper case

Call:
```
procedure ConvUpper(
                  var Str          : PString)
```

Parameters: *Str* - to be converted

Str is converted to upper case. Uses character compares to test whether character is lower case.

CVD

Converts a decimal string to an integer.

Call:
```
function CVD(
                    Str             :PString)
          :integer
```

Parameters: *Str* - is the string to be converted.

Result: An integer containing the value.

Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored.

CVH

Converts an hexadecimal string to an integer.

Call:
```
function CVH(
                    Str             :PString)
          :integer
```

Parameters: *Str* - is the string to be converted.

Result: An integer containing the value.

Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. Lower case 'a'..'f' are the same as upper case 'A'..'F'

CVHS

Converts an integer to a string using hexadecimal radix.

Call:
```
function CVHS(
                    I               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted.

Result: A string containing the character representation; the string will represent the value expressed in hexadecimal.

CVHSS

Converts an integer to a string of width W.

Call:
```
function CVHSS(
                    I               :integer;
                    W               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted

W - is the minimum field width to be produced

Result: A string containing the character representation; the string will be of at least width W, and be filled on the left with spaces and the conversion will be done in hexadecimal radix

The return string is padded on the left with spaces if necessary to fill out the width; radix will be hexadecimal.

CvInt

Converts a string to a integer according to base Radix.

Call:
```
function CvInt(Str: PString;
                        R:  integer)
                    :integer
```

Parameters: *Str* - the string to be converted

Radix - the radix to use

Result: An integer containing the value

Exceptions: *if Overflow then StrBadParm is raised*

Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. A sign is permitted. Lower case 'a'..'z' are the same as upper case 'A'..'Z'

CvL

Converts a string to a long integer.

Call:
```
function CvL(
                Str             : PString;
                Radix           : integer)
        : long
```

Parameters: *Str* - is the string to be converted.

Radix - is the radix to use.

Result: A long integer containing the value.

Converts a string to a long integer according to base Radix Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. A sign is permitted. Lower case 'a'..'z' are the same as upper case 'A'..'Z' Errors: Overflow is possible, but not checked for.

CVN

Exactly the same as CVLS

Call:

```
function CVN(
                    I               : integer;
                    W               : integer;
                    B               : integer;
                    Fill            : Pstring)
            :Pstring
```

CVLS

Converts an integer to a string of width W, padding on the left.

Call:

```
function CVLS(
                    I               : long;
                    W               : integer;
                    Radix           : integer;
                    Fill            : Pstring)
            :Pstring
```

Parameters: *I* - is the integer to be converted

W - is the minimum field width to be produced

B - is the base to use (2..36)

Fill - is a one-character string to fill on the left

Result: A string containing the character representation; the string will be of at least width W, and be filled on the left with Fill, and converted according to radix B

Converts an integer to a string of width W, padding on the left with 'Fill' if necessary to fill out the width. The base for the conversion is Radix. If Radix>10, the letters 'A'..'Z' will be used to compute the character for the representation. Using a base > 36 will produce bogus results. A negative base will force an unsigned result.

CVO

Converts an octal string to an integer.

Call:

```
function CVO(
                    Str             :PString)
            :integer
```

Parameters: *Str* - is the string to be converted

Result: An integer containing the value.

Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored.

CVOS

Converts an integer to a string using octal radix.

Call:
```
function CVOS(
                    I               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted

Result: A string containing the character representation; the string will represent the value expressed in octal

CVOSS

Converts an integer to a string of width W.

Call:
```
function CVOSS(
                    I               :integer;
                    W               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted.

W - is the minimum field width to be produced.

Result: A string containing the character representation; the string will be of at least width W, and be filled on the left with spaces and the conversion will be done in octal radix.

Converts an integer to a string of width W, padding on the left with spaces if necessary to fill out the width; radix will be octal.

CVS

Converts an integer to a string using decimal radix.

Call:
```
function CVS(
                    I               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted.

Result: A string containing the character representation; the string will represent the value expressed in decimal.

CVSS

Converts an integer to a string using decimal radix and fills.

Call:
```
function CVSS(
                    I               :integer;
                    W               :integer)
          :Pstring
```

Parameters: *I* - is the integer to be converted.

W - is the minimum field width to be produced

Result: A string containing the character representation; the string will be of at least width W, and be filled on the left with spaces and the conversion will be done in decimal radix.

Converts an integer to a string of width W, padding on the left with spaces if necessary to fill out the width; radix will be decimal.

CvUp

Returns a copy of Str with lower case replaced by upper.

Call:
```
function  CvUp(
                    Str             : PString)
          : PString
```

Parameters: *Str* - to be converted.

Result: A copy of Str with all alphabetic characters converted to upper case.

Uses character compares to test whether character is lower case.

DeleteChars

This procedure is used to remove characters from a string.

Call:
```
Procedure DeleteChars(
                  var Str               :PString;
                      Index, Size       :Integer)
```

Parameters: *Str* - is the string that is to be changed. Characters will be removed from this string.

Index - is the starting position for the delete.

Size - is the number of character that are to be removed. Size characters will be removed from Str starting at Index.

Result: This procedure does not return a value.

This procedure will change Str. Uses MVBW (Move Bytes Q-code) to copy Str back onto itself.

GetBreak

Allocates and clears a break table.

Call:
```
function GetBreak
         : BreakTable
```

Result: A new, empty break table.

Initial

This function returns true if Str2 is an initial string of Str1.

Call:
```
function Initial(
                    Str1,Str2              : PString)
            :boolean
```

Parameters: *Str1* - is the string to be tested.

Str2 - is the string which is the initial substring to test for.

Result: true if Str2 is an initial substring of Str1.

The comparison is case-sensitive. A null string is an initial substring of any string. Uses EQUBYT QCode to test if the first Length (Str2) chars are equivalent.

InsertChars

Inserts a string into the middle of another string.

Call:
```
Procedure InsertChars(
                    Source          :Pstring;
                var Dest            :PString;
                    Index           :Integer)
```

Parameters: *Source* - is the string that is to be inserted.

Dest - is the string into which the inseration is to be made.

Index - is the starting position, in Dest, for the inseration.

Result: This procedure does not return a value.

Exceptions: *StrLong* - If the resulting string is too long then raise StrLong. Upon resumption the procedure will return a truncated result.

This procedure is used to insert a string into the middle of another string. This procedure will insert Source in Dest starting at location Index.

Lop

Removes and returns the first character from Str.

Call:
```
function Lop(
                    var Str         : PString)
            : PString
```

Parameters: *Str* - is the string from which a character will lopped off.

Result: This function returns the first character of Str.

Removes the first character from Str and returns it as the value of the function. Str no longer

conatains the character. Modifies Str, removes the first character. Uses MVBW to copy the string back onto itself.

Pad

Adds padding characters to a string.

Call:

```
function  Pad(
                Str             : PString;
                TotalLen        : integer;
                PadCh           : char;
                Where           : integer)
        : PString
```

Parameters: *Str* - original string.

TotalLen - desired length for result string.

PadCh - char to insert to achieve desired length.

Where - Where to insert characters; use 0 for padding on left and StrInf (or length(Str)) for padding on the right. If some other value, padding will be done just after the character in that position.

Result: A string of length TotalLen consisting of a copy of Str with sufficient copies of PadCh inserted just after position Where.

Exceptions: *StrBadParm(TotalLen)* - if TotalLen is greater than MaxPStringSize or less than length(Str).

StrBadParm(Where) - if Where is greater than length(Str) or less than 0. If either error resumes, the original value of Str is returned.

Produces a copy of Str adjusted to have length TotalLen by inserting sufficient copies of PadCh just after location Where.

PosC

Finds the position of C in Str. Returns 0 if absent.

Call:

```
function  PosC(
                Str             : PString;
                C               : char)
        : integer
```

Parameters: *Str* - string that is to be searched.

C - character we are looking for.

Result: If C occurs in Str then the index into Str of the first character matching C will be returned. If C was not found (even if Str is empty) then return 0.

If supported, use BSCAN QCode. BreakTable is an array 256 bits long.

PosString

Finds the position of Mask in the substring of S.

Call:
```
function  PosString(
                  Source, Mask          : PString)
          : integer
```

Parameters: *Source* - string that is to be searched.

Mask - pattern that we are looking for.

Result: If Mask occured in Source then the index into (the original) Source of the first character matching the Mask will be returned. If Mask was not found then return 0. If Mask is empty, return 1.

Scans for first character of mask. When found, uses EQUBYT (byte string compare Q-code) to test rest of Mask. The Source is temporarily modified during the search.

ReplaceChars

Replaces a substring of Str with another string.

Call:
```
procedure ReplaceChars(
                  var Str       : PString;
                      NewS      : PString;
                      Index     : integer)
```

Parameters: *Str* - String into which the replacement is to be made.

Index - starting posistion in Str for NewS. If it is greater than Length (Str), no replacement will be made. If it is less then zero, start at one.

NewS - string that is to replace the deleted segment.

Result: The function returns a string of the same length with the appropiate replacement.

Use MVBW QCode to copy NewS over Str.

RevPosC

Tests if c is a member of Str.

Call:
```
Function RevPosC(
                  Str           : PString;
                  c             : char)
          : integer
```

Parameters: *c* - is any char.

Str - is string to test for c member of.

Result: index of c in Str (from end of string) or zero if not there.

RevPosString

Finds the position of Mask in the substring of S.

Call:
```
function  RevPosString(
                Source, Mask            : PString)
        : integer
```

Parameters: *Source* - string that is to be searched.

Mask - pattern that we are looking for.

Result: If Mask occured in Source then the index (from the end) into (the original) Source of the first character matching the Mask will be returned. If Mask was not found then return 0. If Mask is '', return 1.

Scans for first character of mask. When found, uses EQUBYT (byte string compare Q-code) to test rest of Mask. The Source is temporarily modified during the search.

Scan

Scans the string Str according to the breaktable specifications of BT.

Call:
```
function Scan(
                var S           : Pstring;
                    BT          : breaktable;
                var BRK         : Pstring)
        :Pstring
```

Parameters: *S* - is a string to be scanned.

BT - is a breaktable initialized by SetBreak.

BRK - is the break character.

Result: The initial substring determined by the breaktable is removed from Str and returned as the value of the function. The BRK variable contains the string (character) which caused the scan to stop, or the null string if the string was exhausted.

SetBreak

Initializes a break table.

Call:
```
procedure SetBreak(
                var BT                  : BreakTable;
                    Break, Omit         : PString;
                    Options             : BreakKind)
```

Parameters: *Str* - is a string to be scanned.

BT - is a breaktable initialized by SetBreak.

BRK - is the break character.

Result: The initial substring determined by the breaktable is removed from Str and returned as the value of the function. The BRK variable contains the string (character) which caused the scan to stop, or the null string if the string was exhausted.

Exceptions: *StrErrParm* - Illegal combinations of options.

Initializes a breaktable according to the specifications of Break, Omit and Options.

Break specifies the set of characters (as a string) on which a scanning break will occur.

Omit specifies the set of characters which will be removed from the string.

Options allows specification of one option from each of the following groups:

Inclusive	The Break set is the set of characters on which a break will occur.
Exclusive	The Break set is the set of characters on which a break will not occur.
~	If no option is specified from this group, 'Inclusive' is assumed.

Skip	Upon return, the break character will be in the break variable. The result of the scan will be all characters up to the break character, and the input string is modified to start immediately after the break character.
Append	Upon return, the break character will be in the break variable. The result of the scan will be all characters up to and including the break character, and the input string is modified to start immediately after the break character
Retain	Upon return, the break character will be in the break variable. The result of the scan will be all characters up to the break character, and the input string is modified to start at the break character.
~	If no option is specified from this group, 'Skip' is assumed.

FoldUp	Before anything else is done, each character which is alphabetic is folded to uppercase. Note that break sets are

case sensitive, but this is done before the break test. This folding proceeds until the break condition is reached.

FoldDown — Similar to FoldUp, except upper case alphabetics are made lower case.

~ — If no option is specified from this group, no case folding will be done.

*** No guarantees about behavior are made if more than one option is selected from each set group. ***

ShowBreak

Create a string representation of the BreakTable for debugging.

Call:
```
function ShowBreak(
                    BT              : BreakTable)
          : PString
```

Parameters: *BreakTable* - is a breaktable initialized by SetBreak.

Result: An informative string of the break table specifications.

Squeeze

Removes all spaces and tabs from a string .

Call:
```
function Squeeze(
                    Str             : PString)
          : PString
```

Parameters: *Str* - The string to be squeezed.

Result: A string which has all spaces and tabs removed.

Str

This procedure is used to coerce a character to a string.

Call:
```
function Str(
                    Ch              :char)
         :PString
```

Parameters: *Ch* - the character to be coerced to a string.

Result: A string value for a one-character string containing the character.

Strip

Converts sequences of spaces, tabs, CR and LF to a single space.

Call:

```
function Strip(
                    Str             : PString)
          : PString
```

Parameters: *Str* - The string to be squozen.

Result: A string which has all sequences of spaces, tabs, CR and LF changed to a single space.

SubStrFor

Returns a substring

Call:

```
Function SubStrFor(
                    Source          :PString;
                    Index, Size     :Integer)
          :PString
```

Parameters: *Source* - is the string that we are to take a portion of.

Index - is the starting position in Source of the substring.

Size - is the size of the substring that we are to take.

Result: This function returns a substring as described by the parameter list. If Index + Size exceed the dynamic length of the string, return Index to DynamicLength; no error message is generated.

Exceptions: *StrLong* - If Index or Size exceed MaxPStringSize. Upon resumption, this procedure will return a truncated result.

This procedure is used to return a sub portion of the string passed as a parameter.

SubStrTo

Returns a substring

Call:

```
Function SubStrTo(
                    Source                  :PString;
                    Index, EndIndex         :Integer)
          :PString
```

Parameters: *Source* - is the string that we are to take a portion of.

Index - is the starting position in Source of the substring.

EndIndex - is the Ending position in the source of the substring.

Result: This function returns a substring as described by the parameter list. If Index or

EndIndex exceed the dynamic length of the string, return Index to DynamicLength; no error message is generated.

Exceptions: *StrLong* - If Index or EndIndex exceed MaxPStringSize, give an error Upon resumption, this procedure will return a truncated result.

This procedure is used to return a sub portion of the string passed as a parameter.

Trim

Deletes leading and trailing spaces and tabs from a string.

Call:
```
function Trim(
                    Str             : PString)
          : PString
```

Parameters: *Str* - The string to be squozen.

Result: A string which has all leading and trailing spaces and tabs removed.

ULInitial

This function returns true if Str2 is an initial string of Str1.

Call:
```
function ULInitial(
                    Str1,Str2               : PString)
          :boolean
```

Parameters: *Str1* - is the string to be tested.

Str2 - is the string which is the initial substring to test for.

Result: true if Str2 is an initial substring of Str1, false otherwise.

This function returns true if Str2 is an initial string of Str1. The comparison is case-insensitive. A null string is an initial substring of any string.

ULPosString

Find position of a pattern in a string

Call:
```
Function ULPosString(
                    Source, Mask            :PString)
          :Integer
```

Parameters: *Source* - is the string that is to be searched.

Mask

Result: If Mask occured in Source then the index into Source of the first character of Mask will be returned. If Mask was not found then return 0.

This procedure is used to find the position of a pattern in a given string without case sensitivity.

UpChar

Converts C to upper case.

Call:
```
function UpChar(
                C       : Char)
        : char
```

Parameters: *C* - to be converted.

C is converted to upper case. Uses character compares to test whether character is lower case.

UpEQU

Compares two strings for case-independent equality.

Call:
```
Function UpEQU(
                Str1            : PString;
                Str2            : PString)
        :boolean
```

Parameters: *Str1, Str2* - the strings to be compared.

Result: true if the strings are equal, false if they are not, independent of case.

22 Stream package output conversion routines.

Implementers: John Strait

Abstract: This module implements the low-level Pascal I/O. It is not intended for use directly by user programs, but rather the compiler generates calls to these routines when a Reset, Rewrite, Get, or Put is encountered. Higher-level character I/O functions (Read and Write) are implemented by the two modules Reader and Writer.

In this module, the term "file buffer variable" refers to F↑ for a file variable F.

Files: stream.pas

Exported Types

```
SName     = string[255];      same as PathName

FileType = file of Thing
  packed record
   Flag: packed record case integer of
    0: (CharReady   : boolean;      character is in file window
        FEoln       : boolean;      end of line flag
        FEof        : boolean;      end of file
        FNotReset   : boolean;      false if Reset has been performed on this file
        FNotOpen    : boolean;      false if file is open
        FNotRewrite: boolean;       set false if a Rewrite has been performed
                                          on this file
        FExternal   : boolean;      not used - will be permanent/temp file flag
        FBusy       : boolean;      IO is in progress
        FKind       : FileKind);
    1: (skip1      : 0..3;
        ReadError  : 0..7);
    2: (skip2      : 0..15;
        WriteError: 0..3)
   end;
```

Exported Exceptions

```
exception ResetError( FileName: SName );
```

Abstract — Raised when unable to reset a file--usually file not found but also could be ill-formatted name or bad device name.

Parameters — FileName - name of the file or device.

```
exception RewriteError( FileName: SName );
```

Abstract — Raised when unable to rewrite a file--usually file unknown device or partition but also could be ill-formatted name or bad device name.

Parameters — FileName - name of the file or device.

```
exception NotTextFile( FileName: SName );
```

Abstract Raised when an attempt is made to open a non-text file to a character-structured device.

Parameters FileName - name of the device.

```
exception NotOpen;
```

Abstract Raised when an attempt is made to use a file which is not open.

```
exception NotReset( FileName: SName );
```

Abstract Raised when an attempt is made to read a file which is open but has not been reset.

Parameters FileName - name of the file or device.

```
exception NotRewrite( FileName: SName );
```

Abstract Raised when an attempt is made to write a file which is open but has not been rewritten.

Parameters FileName - name of the file or device.

```
exception PastEof( FileName: SName );
```

Abstract Raised when an attempt is made to read past the end of the file.

Parameters FileName - name of the file or device.

```
exception UnitIOError( FileName: SName );
```

Abstract Raised when IOCRead or IOCWrite returns an error status.

Parameters FileName - name of the device.

```
exception TimeOutError( FileName: SName );
```

Abstract Raised when a device times out.

Parameters FileName - name of the device.

```
exception UndfDevice;
```

Abstract Raised when an attempt is made to reference a file which is open to a character-structured device, but the device number is bad. In the current system (lacking automatic initialization of file variables), this may be caused by referencing a file which has never been opened.

```
exception NotIdentifier( FileName: SName );
```

Abstract Raised when an identifier is expected on a file, but something else is encountered.

Parameters FileName - name of the file or device.

```
exception NotBoolean( FileName: SName );
```

Abstract Raised when a boolean is expected on a file, but something else is encountered.

Parameters FileName - name of the file or device.

```
exception BadIdTable( FileName: SName );
```

Abstract Raised by ReadIdentifier when the identifier table is bad.

Parameters FileName - name of the file or device.

```
exception IdNotUnique( FileName: SName; Id: Identifier );
```

Abstract Raised when non-unique identifier is read.

Parameters FileName - name of the file or device. Id - the identifier which was read.

```
exception IdNotDefined( FileName: SName; Id: Identifier );
```

Abstract Raised when an undefined identifier is read.

Parameters FileName - name of the file or device. id - the identifier which was read.

```
exception NotNumber( FileName: SName );
```

Abstract Raised when a number is expected on a file, but something else is encountered.

Parameters FileName - name of the file or device.

```
exception LargeNumber( FileName: SName );
```

Abstract Raised when a number is read from a file, but it is too large.

Parameters FileName - name of the file or device.

```
exception BadBase( FileName: SName; Base: Integer );
```

Abstract Raised when an attempt is made to read a number with a numeric base that is not in the range 2..36.

Parameters FileName - name of the file or device. Base - numeric base (which is not in the range 2..36).

```
exception SmallReal(FileName: SName);

exception LargeReal(FileName: SName);
```

StreamInit

Call:
```
procedure StreamInit(
                var F                           : FileType;
                    WordSize, BitSize           : integer;
                    CharFile                    : boolean )
```

Parameters: *F* - the file variable to be initialized.

WordSize and BitSize - are the size of an element of the file.

CharFile - determines whether or not the file is of characters.

Initializes, but does not open, the file variable F. Automatically called upon entry to the block in which the file is declared. (To be written when the compiler generates calls to it.)

StreamClose

Closes the file variable F.

Call:
```
procedure StreamClose(
                var F           : FileType )
```

Parameters: *F* - the file variable to be closed.

StreamOpen

Opens a file variable.

Call:
```
procedure StreamOpen(
                var F                           : FileType;
                var Name                        : SName;
                    WordSize, BitSize           : integer;
                    CharFile                    : boolean;
                    OpenWrite                   : boolean )
```

Parameters: *F* - the file variable to be opened.

Name - the file name.

WordSize - number of words in an element of the file (0 indicates a packed file).

BitSize - number of bits in an element of the file (for packed files). name.

CharFile - true if the file is a character file.

OpenWrite - true if the file is to be opened for writing (otherwise it is opened for reading).

Exceptions: *ResetError* - if unable to reset the file.

RewriteError - if unable to rewrite the file.

NotATextFile - if an attempt is made to open a non-text file to a character structured device.

Opens the file variable F. This procedure corresponds to both Reset and Rewrite.

GetB

Get the next element of a file.

Call:
```
procedure GetB(
                    var F           : Filetype )
```

Parameters: *F* - the file to be advanced.

Exceptions: *NotOpen* - if F is not open.

NotReset - if F has not been reset.

PastEof - if an attempt is made to read F past Eof.

Advances to the next element of a block-structured file and gets it into the file buffer variable.

GetC

Get the next character of a file.

Call:
```
procedure GetC(
                    var F           : Filetype )
```

Parameters: *F* - the file to be advanced.

Exceptions: *NotOpen* - if F is not open.

NotReset - if F has not been reset.

PastEof - if an attempt is made to read F past Eof.

TimeOutError - if RS: or RSX: times out.

UnitIOError - if IOCRead doesn't return IOEIOC or IOEIOB.

UndfDevice - if F is open, but the device number is bad.

Advances to the next element of a character-structured file and gets it into the file buffer variable.

PutB

Put the next element in a file.

Call:
```
procedure PutB(
                    var F           : Filetype )
```

Parameters: *F* - the file to be advanced.

Exceptions: *NotOpen* - if F is not open.

NotRewrite - if F has not been rewritten.

Writes the value of the file buffer variable to the block-structured file and advances the file.

PutC

Put the next character in a file.

Call:
```
procedure PutC(
                    var F           : FileType )
```

Parameters: *F* - the file to be advanced.

Exceptions: *NotOpen* - if F is not open.

NotRewrite - if F has not been rewritten.

UnitIOError - if IOCWrite doesn't return IOEIOC or IOEIOB.

TimeOutError - if RS: or RSX: times out.

UndfDevice - if F is open, but the device number is bad.

Writes the value of the file buffer variable to the character- structured file and advances the file.

PReadln

Advances to the first character following an end-of-line.

Call:
```
procedure PReadln(
                    var F           : Filetype )
```

Parameters: *F* - the file to be advanced.

PWriteln

Writes an end-of-line.

Call:
```
procedure PWriteln(
                    var F           : Filetype )
```

Parameters: *F* - the file to which an end-of-line is written.

KBFlushBoardOutput

Flushes the typescript output buffer for a file open to 'console:' This is only used by routines in Writer to indicate the end of a "block" of text output.

Call:
```
procedure KBFlushBoardOutput(
```

```
                    var F           : Filetype )
```

Parameters: *F* - the file to be flushed.

StreamKeyBoardReset

Call:
```
procedure StreamKeyBoardReset(
                    var F           : Text )
```

Parameters: *F* - file to be cleared.

Clears the keyboard input buffer and the file variable F so that all input typed up to this point will be ignored.

InitStream

Initializes the stream package. Called by System.

Call:
```
procedure InitStream
```

FullLn

Determines if there is a full line in the keyboard input buffer.

Call:
```
function FullLn(
                    var F           : Text )
          : Boolean
```

Parameters: *F* - file to be checked.

Result: True if a full line has been typed.

Exceptions: *NotOpen* - if F is not open.

NotReset - if F has not been reset.

Determines if there is a full line in the keyboard input buffer. This is the case if a carriage-return has been typed. This function is provided in order that a program may continue to do other things while waiting for keyboard input. If the file is not open to the console, FullLn is always true.

StreamName

Returns the file name associated with the file variable F.

Call:
```
function StreamName(
                    var F           : FileType )
          : SName
```

Parameters: *F* - file variable whose name is to be returned.

Returns the file name associated with the file variable F. For block-structured files, the full path name including device and partition is returned. For character-structured files, the device name is returned. *Environment:* This routine seems to be called by Stream, Reader and Writer when they are

about to raise an exception on the stream and want to know the name or filename associated with the stream. For Accent it gives only the simple filename, which does not include the disk or partition name.

WriteNChars

Writes some number of characters to the file.

Call:

```
procedure WriteNChars(
                    var F          : FileType;
                        c          : char;
                        N          : Integer)
```

Parameters: *F* - the file to write to.

c - the character to duplicate.

N - the number of characters to write.

WriteChars

Writes some number of spaces to the file.

Call:

```
procedure WriteChars(
                    var F          : FileType;
                    var S          : String)
```

Parameters: *F* - the file to write to.

S - the string to write.

IsStreamDevice

Indicates whether a name is one of the special devices that the Stream package uses.

Call:

```
function IsStreamDevice(
                        S          : SName)
              : integer
```

Parameters: *S* - String containing name to check.

Result: Index of the last character of the device name if a stream device, 0 if not a stream device.

23 ViewKern: Graphics operations

Implementers: Brad A. Myers
David Golub

Abstract: This module attempts to call the Kernel protected graphics operations and if they fail, then it calls Sapphire's graphics operations instead. These routines are also documented in **The Spice Programmer's Manual** section **Sapphire Window Manager Procedure Headers.**

Files: viewkern.pas,sapphdefs.pas

Exported Types

```
LineFunct = (DrawLine, EraseLine, XORLine); {used in ViewLine}

RectColorFunct = (RectBlack, RectWhite, RectInvert);used in ViewColorRect

RopFunct = (
        RRpl,     Destination gets source
        RNot,     Destination gets NOT source
        RAnd,     Destination gets Destination AND source
        RAndNot,  Destination gets Destination AND (NOT source)
        ROr,      Destination gets Destination OR source
        ROrNot,   Destination gets Destination OR (NOT source)
        RXor,     Destination gets Destination XOR source
        RXNor);   Destination gets Destination XOR (NOT source)

Viewport = Port;

pVPCharArray = ↑VPCharArray;
VPCharArray = Packed Array[0..1] of Char;

VPStr255 = string[255];
```

VPROP

Does a rasterOp from src to destination using windows

Call:

```
Procedure VPROP(
                destvp          : Viewport;
                funct           : RopFunct;
                dx, dy, width, height: Integer;
                srcVP           : Viewport;
                sx, sy          : Integer)
```

Parameters: *destVP* - the destination viewport. May be same as srcVP.

funct - the rasterOp function.

dx, dy - coordinates of the upper left corner of the rectangle in the destination viewport.

width, height - the width and height of the rectangle to rasterOp.

srcVP - the source viewport. May be same as destVP.

sx, sy - coordinates of the upper left corner of the rectangle in the source viewport.

For setting a rectangle to white or black or inverting a rectangle, call VPColorRect instead of VPROP. Only the displayed portions on the screen are updated. If the dest VP has memory then the covered portions are updated in the offscreen memory. May raise exposed exception if portions in destination are not available in source. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewRop.

VPColorRect

Operates on one rectangle to set, clear or invert all its bits.

Call:

```
procedure VPColorRect(
                vp                      : Viewport;
                funct                   : RectColorFunct;
                x, y, width, height     : Integer);
```

Parameters: *vp* - the viewport to modify.

funct - the operation to do: RectWhite, RectBlack, or RectInvert.

x, y - the upper left corner of the rectangle in vp's coordinate system.

width, height - the width and height of the rectangle to do.

This is more efficient than ViewRop for these operations. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Never generates exposed exception. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewColorRect.

VPScroll

Scrolls a portion of a viewport up, down, left, or right and erases the part that is left.

Call:

```
procedure VPScroll(
                destvp                  : Viewport;
                x, y, width, height,    : Integer
                Xamt, Yamt              : Integer);
```

Parameters: *destvp* - the viewport to modify.

x, y - upper left corner of rectangle's old position with respect to destVP.

width, height - width and height of the area to move.

Xamt - number of bits to move the area horizontally. Negative numbers to move to left, positive to move to right.

Yamt - number of bits to move the area vertically. Negative numbers to move up, positive numbers to move down.

Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewScroll.

VPLine

Draws a line in the viewport clipped to the displayed portions.

Call:

```
procedure VPLine(
                destvp          : Viewport;
                funct           : LineFunct;
                x1,y1,x2,y2     : Integer);
```

Parameters: *destVP* - the viewport to draw the line in.

funct - how to draw the line: DrawLine, EraseLine or XorLine.

x1, y1 - one end of the line. Coordinates are in destVp's coordinate space with 0,0 at the upper left.

x2, y2 - the other end of the line. Both end points are drawn.

Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewLine. BUGS: Due to the current microcode, the line may have holes in it.

VpString

Displays a string in a viewport.

Call:

```
procedure VPString(
                destvp, fontVP          : Viewport;
                funct                   : RopFunct;
            var dx, dy                  : Integer;
            var str                     : VPString255;
                firstCh                 : Integer;
            var lastch                  : Integer);
```

Parameters: *destVp* - the viewport to put the string in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.

funct - the rasterOp function to use when displaying the string.

dx, dy - the starting location for the origin of the first character. Set to be the origin of the next character to be displayed after the characters actually written.

str - the string to display. VAR parameter for efficiency only not modified.

firstCh - the first character of the string to display. If DONTCARE, then 1 is used (first character of the string)

lastch - the last character of the string to display. If DONTCARE, then length(str) is used (the entire string is displayed). Set to the actual last character displayed. This may not be as many characters as was desired because the edge of the viewport was reached.

As much of the string as will fit is displayed and the amount that was displayed is returned. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewString. VPPutString is similar to this procedure but it does not have the return values.

VpCharArray

Displays a portion of a character array in a viewport.

Call:

```
procedure VPChArray(
                destvp, fontVP          : Viewport;
                funct                   : RopFunct;
            var dx, dy                  : Integer;
                chars                   : pVPCharArray;
                arSize                  : Long;
                firstCh                 : Integer
            var lastch                  : Integer);
```

Parameters:

destVp - the viewport to put the string in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.

funct - the rasterOp function to use when displaying the string.

dx, dy - the starting location for the origin of the first character. Set to be the origin of the next character to be displayed after the characters actually written.

chars - Pointer to a packed array of characters that contains the characters to display.

arSize - Total number of characters in the array.

firstCh - the first character of the string to display. If DONTCARE, then 0 is used (first character of the string)

lastch - the last character of the string to display. If DONTCARE, then arSize1 is used (the entire string is displayed). Set to the actual last character displayed. This may not be as many characters as was desired because the edge of the viewport was reached.

Like VPString, except that the characters come from a packed array of characters instead of a string.

VPChar

Displays a single character in a viewport.

Call:

```
procedure VPChar(
                    destvp, fontVP          : Viewport;
                    funct                   : RopFunct;
                var dx, dy                  : Integer;
                    ch                      : Char);
```

Parameters: *destVp* - the viewport to put the character in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP is used.

funct - the rasterOp function to use when displaying the character.

dx, dy - the location for the origin of the character. Set to the origin of the next character to be displayed.

ch - the character to show.

Unlike the other text display routines, this one will not notify the user if the edge of the viewport has been reached. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewChar. VPPutChar is similar to this procedure but it does not have the return value.

VPPutString

Same as VPString except no return values. Displays a string in a viewport.

Call:

```
procedure VPPutString(
                    destvp, fontVP          : Viewport;
                    funct                   : RopFunct;
                    dx, dy                  : Integer;
                    var str                 : VPStr255;
                    firstCh, lastch         : Integer);
```

Parameters: *destVp* - the viewport to put the string in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.

funct - the rasterOp function to use when displaying the string.

dx, dy - the starting location for the origin of the first character.

str - the string to display. VAR parameter for efficiency only not modified.

firstCh - the first character of the string to display. If DONTCARE, then 1 is used (first character of the string)

lastch - the last character of the string to display. If DONTCARE, then length(str) is used (the entire string is displayed).

Displays a string in a viewport. As much of the string as will fit is displayed. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewPutString.

VPPutChArray

Same as VPChArray except no return values. Displays a portion of a character array in a viewport.

Call:

```
procedure VPPutChArray(
                destvp, fontVP          : Viewport;
                funct                   : RopFunct;
                dx, dy                  : Integer;
                chars                   : pVPCharArray;
                arSize                  : Long;
                firstCh, lastch         : Integer);
```

Parameters: *destVp* - the viewport to put the string in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.

functthe rasterOp function to use when displaying the string. - the starting location for the origin of the first character.

chars - Pointer to a packed array of characters that contains the characters to display.

arSize - Total number of characters in the array.

firstCh - the first character of the string to display. If DONTCARE, then 0 is used (first character of the string)

lastch - the last character of the string to display. If DONTCARE, then arSize1 is used (the entire string is displayed).

Like VPPutString, except that the characters come from a packed array of characters instead of a string.

VPPutChar

Same as VPChar except no return values. Displays a single character in a viewport.

Call:

```
procedure VPPutChar(
                destvp, fontVP          : Viewport;
```

```
funct                    : RopFunct;
dx                       : Integer;
dy                       : Integer;
ch                       : Char);
```

Parameters: *destVp* - the viewport to put the character in.

fontVP - a viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP is used.

funct - the rasterOp function to use when displaying the character.

dx, dy - the location for the origin of the character.

ch - the character to show.

Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls Sapphire's ViewPutChar.

24 WindowUtils: Routines to manipulate windows

Implementers: John B. Brodie
David Golub

Abstract: This module provides several useful procedures for manipulating the UserWindow. All of the actions requested by these routines take place on UserWindow. It is assumed that this window has been initialzed and exported by PascalInit. These routines provide a simpler interface to routines documented in **The Spice Programmer's Manual** section **Sapphire Window Manager Procedure Headers.**

Files: windowutils.pas, sapphdefs.pas

Exported Types

```
TitStr              = String[TitStrLength];
TitStrLength        = LandScapeBitWidth div SysFontWidth;
SysFontWidth        = 9;
LandScapeBitHeight  = 1024;
```

ShowPathAndTitle

Display current path and given title in the UserWindow.

Call:
```
Procedure ShowPathAndTitle(
                S               : TitStr)
```

Parameters: *S* - String to be displayed in UserWindow's title line.

This procedure is called to display the current path and the given string in the title line. Assumes that UserWindow is initialized within PascalInit.

ShowWindowErrorFlag

Display ErrorFlag in the UserWindow icon.

Call:
```
Procedure ShowWindowErrorFlag
```

RemoveWindowErrorFlag

Remove the ErrorFlag from the UserWindow icon.

Call:
```
Procedure RemoveWindowErrorFlag
```

ShowWindowRequestFlag

Display the RequestFlag in the UserWindow icon.

Call:
```
Procedure ShowWindowRequestFlag
```

RemoveWindowRequestFlag

Remove the RequestFlag from the UserWindow icon.

Call: `Procedure RemoveWindowRequestFlag`

ShowWindowAttentionFlag

Display the AttentionFlag in the UserWindow icon.

Call: `Procedure ShowWindowAttentionFlag`

RemoveWindowAttentionFlag

Remove the AttentionFlag from the UserWindow icon.

Call: `Procedure RemoveWindowAttentionFlag`

StreamProgress

Shows progress in reading a Pascal File in the title-line progress bar of the UserWindow.

Call:
```
Procedure StreamProgress(
                    var F           : File)
```

Parameters: *F* - The file being read. It must have been Reset and not Closed.

ComputeProgress

Shows progress in the title-line progress bar of the UserWindow, as an amount of a total.

Call:
```
Procedure ComputeProgress(
                    Current, Max            : Long)
```

Parameters: *Current* - How far the operation has gotten.

Max - Total amount for the operation.

RandomProgress

Shows random progress in the title-lin bar of the UserWindow.

Call: `Procedure RandomProgress`

Shows random progress (something is happening, but we're not sure how much) in the title-line progress bar.

QuitProgress

Turns off the title-line progress bar in the UserWindow.

Call: `Procedure QuitProgress`

MultiLevelProgess

Shows progress in the selected progress bar in the UserWindow, as an amount of a total.

Call:

```
Procedure MultiLevelProgress(
                Level           : integer;
                Current, Max    : Long)
```

Parameters: *Level* - Which progress bar to use.

Current - How far the operation has gotten.

Max - Total amount for the operation.

MultiStreamProcess

Shows progress in reading a Pascal file in the selected progress bar of the UserWindow.

Call:

```
Procedure MultiStreamProgress(
                Level           : integer;
            var F               : File)
```

Parameters: *Level* - The progress bar to show progress in.

F - The file being read. It must have been Reset and not Closed.

QuitMultiProgress

Turns off the selected progress bar in the UserWindow.

Call:

```
Procedure QuitMultiProgress(
                Level           : integer)
```

Parameters: *Level* - Which progress bar to use.

A. Error Codes

A.1 Accent

```
General error codes to be used by all modules that pass messages.
Exported by AccentType.

   BADMSGID            = 1;
   WRONGARGS           = 2;
   BADREPLY            = 3;
   NOREPLY             = 4;
   UNSPECEXCEPTION     = 5;  Message is an exception on behalf of a server

Error codes returned by Accent kernel calls. Exported by AccentType.

   AccErr                  = 100;

   Dummy                   = 100;
   Success                 = 101;
   TimeOut                 = 102;
   PortFull                = 103;
   WillReply               = 104;
   TooManyReplies          = 105;
   MemFault                = 106;
   NotAPort                = 107;
   BadRights               = 108;
   NoMorePorts             = 109;
   IllegalBacklog          = 110;
   NetFail                 = 111;
   Intr                    = 112;
   Other                   = 113;
   NotPortReceiver         = 114;
   UnrecognizedMsgType     = 115;
   NotEnoughRoom           = 116;
   NotAnIPCCall            = 117;
   BadMsgType              = 118;
   BadIPCName              = 119;
   MsgTooBig               = 120;
   NotYourChild            = 121;
   BadMsg                  = 122;
   OutOfIPCSpace           = 123;
   Failure                 = 124;
   MapFull                 = 125;
   WriteFault              = 126;
   BadKernelMsg            = 127;
   NotCurrentProcess       = 128;
   CantFork                = 129;
   BadPriority             = 130;
   BadTrap                 = 131;
   DiskErr                 = 132;
   BadSegType              = 133;
   BadSegment              = 134;
```

```
    IsParent                = 135;
    IsChild                 = 136;
    NoAvailablePages        = 137;
    FiveDeep                = 138;
    BadVPTable              = 139;
    VPExclusionFailure      = 140;
    MicroFailure            = 141;
    EStackTooDeep           = 142;
    MsgInterrupt            = 143;
    UncaughtException       = 144;
    BreakPointTrap          = 145;
    ASTInconsistency        = 146;
    InactiveSegment         = 147;
    SegmentAlreadyExists    = 148;
    OutOfImagSegments       = 149;
    NotASystemAddress       = 150;
    NotAUserAddress         = 151;
    BadCreateMask           = 152;
    BadRectangle            = 153;
    OutOfRectangleBounds    = 154;
    IllegalScanWidth        = 155;
    CoveredRectangle        = 156;
    BusyRectangle           = 157;
    NotAFont                = 158;
    PartitionFull           = 159;
```

B. Summary of Calls

The following is a summary of the Pascal library calls. The page on which the operation is fully described appears within square brackets.

Aload

[6] procedure ARunLoad(RunFileName : Path_Name; p : pointer; filesize : long; hiskport : port; LoadDebug : boolean);

[7] procedure ShowRun(p : pointer; MapFileName : Path_Name)

[7] function DateString(date : Internal_Time) : String

[7] function LinkTypeStr(typ : LinkFileType) : string

Bootinfo

CLoad

[11] Function CLoadProcess(FileName : APath_Name; var FileInMem: pointer; var FileSize : long; Proc : Port; LoadDebug: Boolean): GeneralReturn;

Clock

[13] function IOGetTime: long;

CommandParse

[18] procedure InitCmdFile(var InF: pCommand_File_List);

[19] function OpenCmdFile(FileName: pWord_String; var InF: pCommand_File_List) : GeneralReturn

[19] procedure ExitCmdFile(var inF: pCommand_File_List)

[19] procedure ExitAllCmdFiles(var InF: pCommand_File_List)

[20] procedure DstryCmdFiles(var InF: pCommand_File_List)

[20] Procedure InitCommandParse

[20] Procedure DestroyCommandParse

[20] Function ParseCommand(var inputs: pCommand_Word_List; var outputs: pCommand_Word_List; var switches: pCommand_Word_List) : GeneralReturn

[21] Function ParseChPool(ChPool: pCharacter_Pool; PoolLength: Char_Pool_Index; var inputs: pCommand__Word__List; var outputs: pCommand__Word__List; var switches: pCommand_Word_List) : GeneralReturn

[21] Function ExerciseParseEngine (ChPool: pCharacter_Pool; PoolLength: Char_Pool_Index; procedure ReadPool(var Pool: pCharacter_Pool; var PLen: Char_Pool_Index); var inputs: pCommand_Word_List; var outputs: pCommand_Word_List; var switches: pCommand_Word_List) : GeneralReturn;

[22] Function AllocCommandNode(WordClass : Word__Type; WordString: Cmnd__String) : pCommand_Word_List

[22] Procedure DestroyCommandList(var argList: pCommand_Word_List)

[22] Procedure AlwaysEof(var ChPool: pCharacter_Pool; var PoolLength: Char_Pool_Index)

[23] procedure StdError(var table: pWord_Search_Table; CaseSensitive: boolean)

[23] Procedure AddSearchWord(table: pWord__Search__Table; WordKey: integer; WordString: Cmnd_String)

[23] Procedure DeleteSearchWord(table: pWord_Search_table; WordString: Cmnd_String)

[24] Procedure DestroySearchTable(var table: pWord_Search_Table)

[24] Function UniqueWordIndex(table: pWord_Search_Table; ptrWordString: pWord_String; var WordText: Cmnd_String) :integer

[24] procedure ConvertPoolToString(ChPool: pCharacter_Pool; FirstChar: Char_Pool_Index; StringLength: Char_Pool_Index) : Cmnd_String

[25] procedure ConvertStringToPool(CnvStr: Cmnd_String; var ChPool: pCharacter_Pool; var PoolLength: Char_Pool_Index)

[25] procedure DestroyChPool(var ChPool: pCharacter_Pool; var PoolLength: Char_Pool_Index)

[25] Function WordifyPool(ChPool: pCharacter_Pool; PoolLength: Char_Pool_Index; var WordStruct: CommandBlock): GeneralReturn

[26] procedure GetIthWordPtr(i: long; CmndBlock: CommandBlock) : pWord_String

CommandDefs

[28] function Null_CommandBlock: CommandBlock

Configuration

[29] function CF_IOBoard : CF_IOBoardType

[29] function CF_Monitor: CF_MonitorType

[29] function CF_OldZ80 : boolean

[30] function CF_Network: CF_NetworkType

Dynamic

[32] procedure InitDynamic

[32] function CreateHeap : HeapNumber

[32] procedure ResetHeap(S : HeapNumber)

[32] procedure DestroyHeap(S : Heap Number)

[33] procedure DisposeP(var Where : pointer; Len : integer)

[33] procedure NewP(S : HeapNumber; A : integer; var Where : pointer; L : integer)

[34] procedure RaiseP(ES, ER, PStart, PEnd: Integer)

[35] procedure InitExceptions

ExtraCmdParse

[36] Function GetCmd (Prompt : Cmnd_String; SearchTable: pWord_Search_Table; var CmdName: Cmnd_String; var InF: pCommand_File_List; var inputs: pCommand_Word_List; var outputs: pCommand_Word_List; var switches: pCommand_Word_List; var ErrorGR: GeneralReturn): integer;

[38] Function GetShellCmd(SearchTable: pWord_Search_Table; var CmdName: Cmnd_String; var inF: pCommand_File_List; var inputs: pCommand_Word_List; var outputs: pCommand_Word_List; var switches: pCommand_Word_List; var ErrorGR: GeneralReturn): integer

[40] Function GetParsedUserInput(prompt: Cmnd_String; var inF: pCommand_File_List; var inputs: pCommand_Word_List; var outputs: pCommand_Word_List; var switches: pCommand_Word_List) : GeneralReturn

[40] Function GetConfirm (prompt : Cmnd_String; def : integer; var switches: pCommand_Word_List) : integer

[41] Procedure GetCharacterPool(prompt: Cmnd_String; var InputFile: Text; var ChPool: pCharacter_Pool; var PoolLength: Char_Pool_Index)

IPCRecordIO

[43] function SendRecord(localport : Port; remoteport : Port; id : long; MsgType : long; recptr : Pointer; recsize : integer) : GeneralReturn;

[43] function RecRecord(var localport : Port; var remoteport : Port; var id : long; var MsgType : long; var recptr : Pointer; var recsize : integer) : GeneralReturn;

OldTimeStamp

[45] function OldCurrentTime: TimeStamp;

[45] function NewToOldTime(NewTime: Internal_Time):TimeStamp;

[45] function OldToNewTime(OldTime: TimeStamp): Internal_Time;

PascalInit

[47] Procedure InitPascal

[47] Procedure InitPascal (AmIClone : BOOLEAN)

[47] Function DisablePrivs(Proc: PORT): GeneralReturn;

[48] Function EnablePrivs(Proc: PORT): GeneralReturn;

PathName

[50] function ReadFile(Var PathName : Path_Name; Var Data : File_Data; Var ByteCount : long) : GeneralReturn;

[51] function ReadExtendedFile(Var PathName : Path_Name; ExtensionList : Extension_List; ImplicitSearchList : Env_Var_Name; Var Data : File_Data; Var ByteCount : long) : GeneralReturn;

[52] function WriteFile(Var PathName : Path_Name; Data : File_Data; ByteCount : long) : GeneralReturn;

[52] function CompletePathName(var WildPathName : Wild_Path_Name; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean; var Cursor : integer) : long;

[53] function ExpandPathName(Var PathName : Wild_Path_Name; ImplicitSearchList : Env_Var_Name) : GeneralReturn;

[54] function FindPathName(Var PathName : Path_Name; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean; Var EntryType : Entry_Type; Var NameStatus : Name_Status) : GeneralReturn;

[55] function FindFileName(Var PathName : Path_Name; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean) : GeneralReturn;

[55] function FindExtendedPathName(Var PathName : Path_Name; ExtensionList : Extension_List; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean; Var EntryType : Entry_Type; Var NameStatus : Name_Status) : GeneralReturn;

[56] function FindExtendedFileName(Var PathName : Path_Name; ExtensionList : Extension_List; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean) : GeneralReturn;

[57] function FindTypedName(Var PathName : Path_Name; ExtensionList : Extension_List; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean; Var EntryType : Entry_Type; Var NameStatus : Name_Status) : GeneralReturn;

[58] function FindWildPathnames(Var WildPathName : Path_Name; ImplicitSearchList : Env_Var_Name; FirstOnly : boolean; NameFlags : Name_Flags; EntryType : Entry_Type; Var FoundInFirst : boolean; Var DirName : APath_Name; Var EntryList : Entry_List; Var EntryListCnt : long) : GeneralReturn;

[59] procedure ExtractSimpleName(Name : Path_Name; Var StartTerminal : integer; Var StartVersion : integer);

[60] function SimpleName(PathName : Path_Name) : Entry_Name;

[60] function StripCurrent(Var WildPathName : Wild_Path_Name) : GeneralReturn;

[60] Procedure AddExtension(Var FileName : Path_Name; Extension : String);

[61] Procedure ChangeExtensions(Var Name : Path_Name; EList : Extension_List; NewExt : string);

[61] function NextExtension(Var EList : Extension_List) : string;

[61] Procedure RemoveExtension(Var FileName : Path_Name; Extension : String);

[62] function Index1Unquoted(S : Wild_Path_Name; C : char) : integer;

[62] function IsQuotedChar(S : Wild_Path_Name; Index : integer) : boolean;

PMatch

] procedure PattDebug(v : boolean)

[63] function IsPattern(str : pms255) : boolean

[64] function PattMatch(var str, pattern : pms255) :boolean

[64] function PattMap(var str,inpatt,outpatt,outstr :pms255; fold :boolean) :boolean

RealFunc

[66] function Sqrt(X : Real): Real

[66] function Ln(X : Real): Real

[66] function Log10(X : Real): Real

[66] function Exp(X : Real): Real

[66] function Power(X, Y : Real): Real

[67] function PowerI(X : Real; Y : Integer) : Real;

[67] function Sin(X : Real): Real

[67] function Cos(X : Real): Real

[67] function Tan(X : Real): Real

[67] function CoTan(X : Real): Real

[68] function ArcSin(X : Real): Real

[68] function ArcCos(X : Real): Real

[68] function ArcTan(X : Real): Real

[68] function ArcTan2(Y, X : Real): Real

[69] SinH(x: real) : real

[69] function CosH (x: real) :real

[69] function TanH(x: real) : real

SaltError

[70] Procedure GRWriteStdError(GR : GeneralReturn; ER_Type : GR_Error_Type; InMsg : PString)

[70] Procedure GRStdError(GR : GeneralReturn; ER_Type : GR_Error_Type; InMsg : PString; var OutMs : PString)

[71] Procedure GRWriteErrorMsg(GR : GeneralReturn; ER_Type : GR_Error_Type; ProgName : String; InMsg : PString)

[71] Procedure GRErrorMsg(GR : GeneralReturn; ER_Type : GR_Error_Type; ProgName : String; InMsg : PString; var OutMsg : PString)

[72] Procedure ErrorMsgPMBroadcast(GR : GeneralReturn; ER_Type : GR_Error_Type; ProgName : String; InMsg : PString)

[72] Function GRStdErr(GR : GeneralReturn; ER_Type : GR_Error_Type; InMsg : PString; var OutMsg : PString) : boolean;

Spawn

[73] function Exec(VAR ChildKPort : Port; VAR ChildDPort : Port; ProcessName : STRING; HisCommand : CommandBlock) : GeneralReturn;

[74] function Split(var ChildKPort : Port; var ChildDPort : Port) : GeneralReturn;

[74] function Spawn(var ChildKPort : Port; var ChildDPort : Port; ProgName : APath - Name; ProcName : string; HisCommand : CommandBlock; DebugIt : boolean; ProtectChild : boolean; SapphConn : ConnectionInheritance; pWindow : Port; pTypeScript : Port; EMConn : ConnectionInheritance; pEMPort : Port; PassedPorts : ptrPortArray; NPorts : long; LoaderDebug : BOOLEAN) : GeneralReturn;

Spice_String

[78] procedure Adjust(var Str :PString; Len :Integer)

[78] procedure AppendChar(var Str : PString; c : Char)

[78] procedure AppendString(var Str1 : PString; Str2 : PString)

[79] function Cat3(Str1,Str2,Str3 : PString) : PString

[79] function Cat4(Str1,Str2,Str3,Str4 : PString) : PString

[79] function Cat5(Str1,Str2,Str3,Str4,Str5 : PString) : PString

[80] function Cat6(Str1,Str2,Str3,Str4,Str5,Str6 : PString) : PString

[80] function Concat(Str1,Str2 : PString) : PString

[80] procedure ConvUpper(var Str : PString)

[81] function CVD(Str :PString) :integer

[81] function CVH(Str :PString) :integer

[81] function CVHS(I :integer) :Pstring

[81] function CVHSS(I :integer; W :integer) :Pstring

[82] function CvInt(Str: PString; R: integer) :integer

[82] function CvL(Str : PString; Radix : integer) : long

[83] function CVN(I : integer; W : integer; B : integer; Fill : Pstring) :Pstring

[83] function CVLS(I : long; W : integer; Radix : integer; Fill : Pstring) :Pstring

[83] function CVO(Str :PString) :integer

[84] function CVOS(I :integer) :Pstring

[84] function CVOSS(I :integer; W :integer) :Pstring

[84] function CVS(I :integer) :Pstring

[84] function CVSS(I :integer; W :integer) :Pstring

[85] function CvUp(Str : PString) : PString

[85] Procedure DeleteChars(var Str :PString; Index, Size :Integer)

[85] function GetBreak : BreakTable

[86] function Initial(Str1,Str2 : PString) :boolean

[86] Procedure InsertChars(Source :Pstring; var Dest :PString; Index :Integer)

[86] function Lop(var Str : PString) : PString

Stream

[101] function FullLn(var F : Text) : Boolean

[101] function StreamName(var F : FileType) : SName

[102] procedure WriteNChars(var F : FileType; c : char; N : Integer)

[102] procedure WriteChars(var F : FileType; var S : String)

[102] function IsStreamDevice(S : SName) : integer

ViewKern

[103] Procedure VPROP(destvp : Viewport; funct : RopFunct; dx, dy, width, height: Integer; srcVP : Viewport; sx, sy : Integer)

[104] procedure VPColorRect(vp : Viewport; funct : RectColorFunct; x, y, width, height : Integer);

[104] procedure VPScroll(destvp : Viewport; x, y, width, height, : Integer Xamt, Yamt : Integer);

[105] procedure VPLine(destvp : Viewport; funct : LineFunct; x1,y1,x2,y2 : Integer);

[105] procedure VPString(destvp, fontVP : Viewport; funct : RopFunct; var dx, dy : Integer; var str : VPString255; firstCh : Integer; var lastch : Integer);

[106] procedure VPChArray(destvp, fontVP : Viewport; funct : RopFunct; var dx, dy : Integer; chars : pVPCharArray; arSize : Long; firstCh : Integer var lastch : Integer);

[107] procedure VPChar(destvp, fontVP : Viewport; funct : RopFunct; var dx, dy : Integer; ch : Char);

[107] procedure VPPutString(destvp, fontVP : Viewport; funct : RopFunct; dx, dy : Integer; var str : VPStr255; firstCh, lastch : Integer);

[108] procedure VPPutChArray(destvp, fontVP : Viewport; funct : RopFunct; dx, dy : Integer; chars : pVPCharArray; arSize : Long; firstCh, lastch : Integer);

[108] procedure VPPutChar(destvp, fontVP : Viewport; funct : RopFunct; dx : Integer; dy : Integer; ch : Char);

WindowUtils

[110] Procedure ShowPathAndTitle(S : TitStr)

[110] Procedure ShowWindowErrorFlag

[110] Procedure RemoveWindowErrorFlag

[110] Procedure ShowWindowRequestFlag

[111] Procedure RemoveWindowRequestFlag

[111] Procedure ShowWindowAttentionFlag

[111] Procedure RemoveWindowAttentionFlag

[111] Procedure StreamProgress(var F : File)

[111] Procedure ComputeProgress(Current, Max : Long)

[111] Procedure RandomProgress

[111] Procedure QuitProgress

[112] Procedure MultiLevelProgress(Level : integer; Current, Max : Long)

[112] Procedure MultiStreamProgress(Level : integer; var F : File)

[112] Procedure QuitMultiProgress(Level : integer)